PG iTraffic with TurtleBots: Project Report

Debkowski, Julia
Eilers, Nellson
Gerber, Stefan
Grave, Malte
Heckelmann, Lasse
Kowalska, Paulina
Marken, Marie
Monenschein, Jan-Magnus
Schneiders, Carl
Struck, Simon
Wojciak, Filip

April 3, 2024

Contents

Chapter 1

Product Vision

The goal of the project group iTraffic with TurtleBots is to develop autonomous driving functions with TurtleBot (TB)s to solve scenarios of varying complexity based on a German autobahn without curves. To assure that these driving functions meet their specifications, test-based validation is used.

The project group uses the TB platform as a basis. This makes it possible to experimentally validate autonomous driving functions in different simulated traffic scenarios with the use of mostly low-cost sensor technology. The goal is to present and tackle the challenges of autonomous driving in a way that is cost-effective and low-risk.

In order to achieve this, a platform for creating autonomous driving functions based on the TB is developed. It enables members of the project group as well as future developers to implement controllers for vehicles on different autonomy levels and simulating those vehicles on a TB. This platform is called TurtleCar.

Additionally, TurtleCar provides capabilities to validate the controllers in a simulated as well as a real life environment. For this, a Domain Specific Language (DSL) to define test cases for the controllers is developed. The simulation suite „Gazebo“ is used for testing in a simulated environment. Using the simulation, it is possible to automatically execute test cases, gather the results and determine whether the test conditions where met. TurtleCar ensures that both environments behave similarly with regard to the inputs and outputs of the controller.

Using the TurtleCar platform, several scenarios of various complexities are developed and provided, and from those, test cases for the testbed are derived. This enables a developing cycle that is closely related to the DevOps method: The development of the testbed follows the requirements posed by the scenarios, and can be adjusted as needed.

1.1 Autonomy Levels

The scenarios developed as part of this project group are all be based on a highway with the properties of a German autobahn without curves. Controllers with three different levels of autonomy are built:

- no autonomy

- partially automated
- highly automated

The driving functions used by these levels are implemented using suitable, robust control strategies. They are based on the current state of the art in the control engineering domain.

1.2 Functions of the Non-Autonomous Vehicle

The non-autonomous vehicle has no assistance systems. In the non-autonomous vehicle, a human driver controls the vehicle completely. They can define the speed and the steering angle, and the bot moves according to the vehicle's dynamics.

1.2.1 Functions of the Partially Automated Vehicle

The partially automated vehicle can perform certain functions autonomously within bounded conditions. It may call for the driver's intervention if needed.

Lane Keeping Assistant The driver determines the speed of the vehicle. As long as the Lane Keeping Assistant (LKA) is activated, the vehicle keeps to the center of its current lane without the need of the driver to control the steering angle.

Adaptive Cruise Control The maximum speed of the vehicle is determined by the driver. When Adaptive Cruise Control (ACC) is activated, and another, slower vehicle is driving in the front, the speed is adjusted so that a safety margin is kept.

Lane Changing When the Lane Changing function is engaged, if conditions permit, the vehicle executes lane changes, while maintaining appropriate spacing from neighboring vehicles.

Collision Avoidance System The vehicle avoids static obstacles like road works by changing lanes or stopping safely before the obstacle until a safe lane changing is possible.

Overtaking The vehicle avoids obstacles moving in the same lane, like a slower car ahead, by changing lanes or reducing speed until a safe lane changing is possible.

1.2.2 Functions of the Highly Automated Vehicle

The highly automated vehicle is able to drive on the highway without needing intervention from the driver. All actions are self-initiated. Using only the aforementioned driving functions, it moves the vehicle forward as safely as possible without any input from the driver while adhering to the German traffic regulations in terms of safety margins. Also, it adheres to speed limits and „no overtaking“ road signs.

Malicious Agent Avoidance The vehicle avoids collisions with cars which are moving in defiance of traffic rules by choosing a safe driving strategy.

Platooning In platooning mode, the vehicle joins a closely coordinated group of vehicles traveling in a convoy-like formation. The system automatically controls the vehicle's speed, following distance, and positioning within the platoon. The platooning system continuously communicates with other vehicles in the group, ensuring safe and efficient travel.

Chapter 2

State of the Art

This section introduces the autonomy levels according to the Society of Automotive Engineers (SAE), shows the group's ongoing research on driving functions, and outlines past projects working on similar topics.

2.1 Levels of Automotive Autonomy

The SAE defines levels of autonomy in on-road automated driving vehicles. The SAE standard J3016_202104 [86] outlines the six levels of driving automation, ranging from Level 0 (no automation) to Level 5 (full automation) depicted in Table 2.1 as follows.

Table 2.1: Levels of driving automation according to the SAE standard J3016_202104 [86]

Level	Description
Level 0	No Driving Automation
Level 1	Driver Assistance
Level 2	Partial Driving Automation
Level 3	Conditional Driving Automation
Level 4	High Driving Automation
Level 5	Full Driving Automation

While level $1-2$ use „driver support" features, level $3-5$ use „automated driving" features. The level of driving automation of a vehicle is determined by a combination of factors: the extent of required human involvement in driving tasks, the vehicle's capability to perform driving functions, and the operational design domain under which a feature is designed to function (i. e. environmental restrictions). The standard also differentiates between three types of actors: the (human) user, the driving automation system, and other vehicle systems and components.

Because of this, systems that provide alerts about driving hazards are excluded from this classification as they neither automate driving tasks nor change the driver's role in performing them. Additionally, the LKA, the electronic stability control or other certain types of driver assistance systems are not covered

by this driving automation classification. This is because it provides momentary intervention rather than sustained automation of driving tasks.

2.2 Related Projects

In the past, there were several projects from the Carl von Ossietzky University of Oldenburg who dealt with implementing driving functions on hardware representing a vehicle. In the following, these will be described and distinguished from the project group.

„Realtime Controlled Cooperative Autonomous Racing System“ (RCCARS) has undertaken the task to develop a safety-critical system using the racetrack Mini-Z Grand Prix Circuit 30 and RC-Cars from Kyosho. This system is responsible for observing and controlling autonomously operating vehicles on a racetrack. In their „collision-free“ scenario, a single car is supposed to autonomously complete five laps on the racetrack at a minimum average speed of $1.5\,^{\mathrm{m}}/_{\mathrm{s}}$ without colliding with the track's boundaries [11].

„Realtime Controlled Cooperative Autonomous Racing System Next Generation“ (RCCARSng) builds upon the work of RCCARS. It extends the project by adding a second car and several static obstacles. Both cars are supposed to complete a minimum of ten collision-free laps. During this, both vehicles have the opportunity to overtake each other and should avoid obstacles while doing so. This group divides their scenario „collision-free overtaking“ in the following three variants [9].

- One vehicle following the other.
- One vehicle overtaking the other.
- Following and overtaking while avoiding obstacles.

RCCARS and RCCARSng both use global knowledge and external calculations. A camera situated above the racetrack perceives the track and the vehicles on the track. There exists an external component responsible for location determination and for controlling the vehicles. For the overtaking function, they use a preceding trajectory calculation implemented in Matlab.

„Emergency Braking Assistant for fully Autonomous Cars“ (EmBrAAC) has undertaken the task to develop a real-time vehicle assistant. Depending on the situation, it should be capable of calculating an evasive strategy or performing emergency braking. They use a remotely-controlled vehicle from Traxxas in combination with a predefined and self-build course. Their focus lies on real-time capabilities and contract-based design [23].

Within the context of the university course „Forschendes Lernen - Mobiles Multiagenten-Robotersystem“ eight students investigated and practically implemented method-oriented topics in the field of mobile robotic systems using a TB. The course was meant as a preliminary project for the „iTraffic with TurtleBots“ project group and was attended by some people from this project group. They familiarized themselves with the simulation software Gazebo and used it to validate initial prototypes before transferring them into real hardware. After doing some fundamental work with the TB and Gazebo software, the students were split into two groups.

One group focused on using Simulink to address the question „How can an autonomous driving function for obstacle avoidance be developed?“. As part of this, they developed control algorithms that enable the robot to follow the desired path, navigate around obstacles, and perform precise navigation.

The other group, using Python, explored the question „How is realistic driving behavior simulated?“. In doing so, they researched vehicle models and implemented a suitable one. This included considering factors such as friction, inertia, road conditions, and other physical properties.

During the course, Simulink and Python were compared for the implementation of driving functions on a TB. The project group adopted the Mathematical Vehicle Model (MVM) and knowledge about the differences between reality and Gazebo simulation.

In comparison to these related projects, the project group „iTraffic with TurtleBots“ enables the utilization and implementation of driving functions on a TB based on local knowledge. The implemented driving functions use a camera und a LIDAR sensor on the TB. These sensors can be combined freely. The environment in which the TB operates and the TB itself closely resemble reality: The TB is located on a three-lane highway and behaves like a specific car. The goal is to develop a modular development platform. That means vehicle models, environments and driving functions can be added and are interchangeable. Alongside the creation of the development platform, an automated testing platform is created. This allows experimentally validating the driving functions.

Chapter 3

Used Software and Hardware

The creation of the project was conducted using various software such as Gazebo and ROS2. Hardware such as the TB and its eqquiped sensors was also employed. This chapter gives brief insights into the used hardware and software.

3.1 TurtleBot3

The TurtleBot3 Burger is a popular and versatile open-source robot platform designed for education, research, and hobbyist applications. Developed by ROBOTIS, it provides a low-cost, customizable solution for learning and experimenting with robotics. The TB consists of two computational boards: a Raspberry Pi 4 and the OpenCR Board. In addition, the TB is equipped with different sensors [93].

Raspberry Pi 4 The Raspberry Pi 4 serves as the host on which a realtime Linux based operating system is running. The entire network communication is executed on the Raspberry Pi 4, this also includes components from the software stack [71].

OpenCR 1.0 The OpenCR board is an open source board which is used to control the actuators and read sensors from the board. It is also possible to connect additional sensors directly. The entire voltage of the system is also managed via this board. The firmware is based on the Arduino framework and can be customized by any needs [91].

The TB is delivered with differently equipped sensors. The main sensors are the LIDAR (Light Detection and Ranging) and a Raspberry camera.

LIDAR The LIDAR is not directly connected to the OpenCR board. It is indeed connected to the Raspberry Pi 4 over a RS232 transducer board, which itself is connected to the USB port of the Raspberry Pi 4. There are different versions of the TB LIDAR, the used TB has the LIDAR LDS-2 equipped [90].

Camera The used camera is an RASP CAM 3, which is directly connected to the CSI (Camera Serial Interface) port of the Raspberry Pi 4. It features a 12-MP image sensor and a max resolution of $1080p$@50Hz [70].

3.2 Additional Sensors

The integration of various additional sensors such as a GPS sensor, a compass, and a temperature sensor greatly enhances the capabilities and functionality of a TurtleBot3, making it suitable for a wide range of scenarios in different environments. The additional sensors were purchased in order to describe further scenarios, but since integrating was not a priority they are currently unused.

3.2.1 GPS Sensor

The GPS sensor on a TurtleBot3 enables precise localization and navigation in outdoor environments. Here are some scenarios where a GPS sensor proves invaluable:

Autonomous Navigation The GPS sensor allows the TurtleBot3 to navigate autonomously in outdoor spaces by providing position information. This is particularly useful in scenarios where field monitoring is required.

Mapping and Exploration With GPS data, the TurtleBot3 can create maps of outdoor environments.

3.2.2 Compass

The compass sensor provides orientation information, allowing the TurtleBot3 to maintain direction and navigate accurately. Here's how it can be utilized:

Heading Control In scenarios where maintaining a specific heading is crucial, the compass sensor ensures the TurtleBot3 stays on course even in the absence of visible landmarks or GPS signal, providing reliable navigation in challenging environments.

Calibration Assistance The compass sensor aids in calibration procedures, ensuring the accuracy of other sensors onboard the TurtleBot3, particularly in scenarios where precise sensor fusion is required for navigation or mapping applications.

3.2.3 Temperature Sensor

The temperature sensor provides environmental data crucial for various applications. Here's how it can be utilized:

Environmental Monitoring By measuring ambient temperature, the TurtleBot3 can monitor environmental conditions in real-time, so scenarios can be created for different temperatures, such as driving on an icy road.

In conclusion, the integration of a GPS sensor, a compass, and a temperature sensor could expand the capabilities of a TurtleBot3, enabling it to operate in diverse scenarios.

3.3 Gazebo

Gazebo is a free and open-source simulation tool that provides the ability to efficiently simulate robots in custom indoor and outdoor environments. It offers a robust physics engine and the ability to simulate various sensors and actuators.

Gazebo plays a critical role in testing and refining the algorithms for controlling TurtleBots. The primary advantage of using *Gazebo* lies in its ability to create realistic simulations that closely mimic the real world. This means the project group could experiment with various scenarios and conditions that the TurtleBots might encounter without needing access to large physical spaces for testing. The creation of the project was conducted.

3.4 ROS 2

Robot Operating System (ROS) provides a framework for communicating between different components based on the Data Distribution Service (DDS), which provides reliable communication in multi-robot systems [52]. The communication is organized by creating nodes for each communication participant which communicate over specified channels called topics. The messages that can be sent over each topic are well-defined and known to all communication participants. ROS also provides checking reliability properties of network members by annotating each message with timestamps and defining properties to topics in order to prevent messages to be exchanged with an undesirable delay or message loss. The system can be used within one robot to coordinate different sensors and actuators as well as between different ROS nodes that can reach each other over the same IP network [52].

A ROS network of several nodes and topics can be seen in XYZ. Here, the Node A publishes messages Twist, which represent speed and angular velocity in three dimensions each, to a topic cmd_vel. The second Node B subscribes to that topic and receives each message published to it.

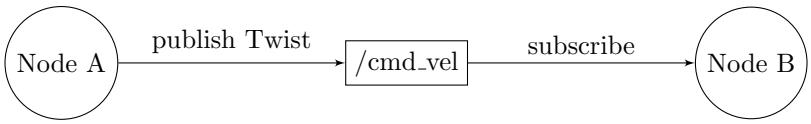


Figure 3.1: A simple ROS 2 network with two nodes communicating over a topic using Twist messages.

Since the communication via topics is the method chosen for this project, other features of ROS such as services and actions are not further discussed here.

Chapter 4

Reflecting Reality

The driving functions to be developed and the used environment should be as realistic as possible to allow accurate emulation of vehicles. For this purpose, the environment has to be defined in a way that makes it usable in reality and in a Gazebo simulation. This is done by scaling down the real environment, the autobahn and its participants, to TB dimensions, which is described in detail in this section. Furthermore, the safety measures employed should follow those used on a German autobahn and be scaled accordingly. The following terminology is defined for disambiguation.

Environment The scaled down environment used by TurtleCar. When prefixed with **real**, specifically the real environment made of wooden panels and cardboard is referenced. Respectively, the **simulated** environment targets the Gazebo simulation.

Road Model The actual, real road parameters that are the origin of the scaled down environment, i. e. the German highway.

Scaling Factor The factor by which a road model distance unit is scaled down to the environment distance.

4.1 Real Environment

In this section, the technical details of the real counterparts used to create the scaled down environment are described. More precisely, the TB 3 Burger model, the Golf VII car model, and the road model of a German autobahn are considered. These parameters are referenced in Section 4.3, where their values are used to construct the scaled down environment.

4.1.1 TurtleBot 3 Burger

To emulate a car on the TB the technical details of the TB model are needed. Even though a TB 3 Specifications Guide exists, the technical parameters were

determined empirically by the project group, to make a precise comparison between reality and specifications possible. The gathered data is depicted in Table 4.1. The data from the specifications is depicted in Table 4.2. Please note that this information does not apply when using the TB in the Gazebo environment.

The value column corresponds with the value which is provided through the `/cmd_vel` topic. Anything below or above the min / max values will be ignored by the TB. Position / Speed accuracy for the odometer was not collected because the `/odom` topic already returns a covariance matrix indicating the accuracy of the measurement.

Table 4.1: Experimentally determined TB 3 Burger parameters

Parameter	**Value(s)**	**Notes**
Velocity	$[0.01, 0.22)$	
Velocity Increment	0.01	
Turn Velocity	$[0.01, 2.64)$	
min. Turn Velocity (unreliable)	0.01	irregular speed, stuttering
min. Turn Velocity (reliable)	0.1	
Turn Velocity Increment	0.01	not 100% certain

The specification defines the following technical parameters for the TB 3 Burger model [73]:

Table 4.2: TB 3 parameters from specification

Parameter	**Value**
Max. Velocity	$0.22\,{}^{\mathrm{m}}/_{\mathrm{s}}$
Max. Turn Velocity	$2.84\,{}^{\mathrm{rad}}/_{\mathrm{s}}$ ($162.72\,{}^{\mathrm{deg}}/_{\mathrm{s}}$)
Size (Length, Width, Height)	138 mm, 178 mm, 192 mm

The minimal speed increment of $0.01\,{}^{\mathrm{m}}/_{\mathrm{s}}$ poses a problem for the velocity calculations inside the Transposer component of TurtleCar Core. The problem is that the transposer is calculating a new TB velocity every time step to simulate a correct acceleration. If the calculated velocity step for a given time step is smaller than $0.01\,{}^{\mathrm{m}}/_{\mathrm{s}}$ then the TB doesn't change its velocity for the current time step. The current velocity v_k is used to calculate the velocity for the next time step v_{k+1} using the formula:

$$v_{k+1} = v_k + a_k \cdot T$$

Where a_k is the current acceleration and T is the length of the time step. For a constant T and very lowa_k, this would conclude that the TB would not increase in speed. To counteract this issue, the transposer is saving the remaining velocity part v_r which is not handled by the TB until the next time step and adds this part to the velocity calculation.

$$v_{k+1} = v_k + v_r + a_k \cdot T$$

This ensures that after enough time steps the TB will reach a velocity step greater than $0.01{}^{\mathrm{m}}/_{\mathrm{s}}$, that can be executed.

4.1.2 Golf VII

Scaling the road model down to the representative environment will be primarily done using a scaling factor derived from the size comparison between the TB and a real VW Golf VII. The Table 4.3 depicts the required specifications of said car model [87].

Table 4.3: Golf VII parameters

Parameter	**Value**
Max. Velocity	69,44 $^{m}/_{s}$ / 250 $^{km}/_{h}$
Size (Length, Width, Height)	4287 mm, 1789 mm, 1478 mm

4.1.3 Road Model

The used road model is based on a standard German highway. In general, the measurements given in the Table 4.4 are used [85].

Table 4.4: German highway dimensions

Parameter	**Value**
Lane Width	2.75 m - 3.75 m
Dash Mark Width	normal 15 cm, broader 30 cm
Dash Mark Length	6 m
Dash Mark Spacing	12 m

4.2 Scaling Approaches

Initially the TurtleCar platform used only a single-factor scaling across all dimensions. This is unsatisfactory, since the TB differs in its aspect ratio regarding a normal car.

If the TB should emulate a real car, its environment must be scaled down in a consistent manner. The most basic scaling strategy sets the TB's width, height and top speed in relation to those of a reference car, like the Golf VII. This results in the scale factors depicted in Table 4.5. This strategy is called „Three Factor Scaling Strategy“ in the following.

However, this also results in three different scaling factors - which is unintuitive and hard to follow for the development of driving functions. It would essentially create three dimensions, with different relations between them. The calculations would be made harder than necessary, and the goal of providing a simple-to-use platform for developing driving functions would not be achieved.

Therefore, other scaling strategies have been developed in the project group, which aim to provide similar results, but result in fewer amounts of scaling factors. They are described in the following.

Table 4.5: Three Factor Based Scaling

Parameter	Value	Unit
Width TB	0.178	m
Width Golf VII (incl. Mirror)	2.073	m
Width Scale (X-Axis)	**0.086**	
Length TB	0.178	m
Length Golf VII	2.073	m
Length Scale (Y-Axis)	**0.086**	
Max Speed TB	0.178	m
Presumed Max Speed Golf VII	2.073	m
Speed Scale	**0.086**	

Different Car This strategy would use the same three factors as above, but compare the width, height and speed to a car which is closer in dimensions to the TB. The assumption is that this would still result in different scaling factors per dimension, but since they would be closer together, they would not differ in relation to that of the TB quite as much, providing a more intuitive scaling at least.

Change Maximum Emulated Speed For this, width and length factors are calculated like before. However, the max speed of the considered car model is artificially limited. This can manipulate the speed factor to be equal to the length scale factor, keeping the number of scaling factors down at two.

Bot-As-Is This strategy essentially keeps the TB as it is. The approach of scaling the TB to the dimensions of a real car would be discarded, subverting the aspect of emulating a „real“ environment.

Width-Speed-Scaling This strategy calculates the same width and speed scale factors as the „Three Factor Scaling Strategy“. The scaling factor for length is set to be equal to the width scale factor. This results in a distortedly scaled TB which is very short in theory, but it also employs only two scaling factors.

4.2.1 Settling on a Strategy

First, the „Different Car“ is neglected since choosing a reference car that has a very common aspect ratio is important to be able to emulate different cars. A very specific but effectively rare car that has an aspect ratio closer to the TB does not fit this reasoning. Also, the resulting factors were still not identical and could have lead to inconsistencies. Second, the „Change Maximum Emulated Speed“ would limit the maximum emulated speed to 22.35 $^{km}/_{h}$, which is not feasible for the project group's goals. Third, the „Bot-As-Is“ would just ignore the problem entirely, essentially eliminating any goals for realistic scenarios and emulating of vehicles. Because of these considerations, these strategies were eliminated and are not considered further.

The „Width-Speed-Scaling Strategy“ proved to be the most promising, since it would result in two factors only and its consequences seemed manageable. Its disadvantages are discussed further in the Subsection 4.2.2.

Calculating the Factors of Width-Speed-Scaling The first factor is based on the ratio of the width of a Golf VII to that of a TB. The width of a Golf VII, which is approximately 2.073 m, is set in relation to the width of the TB, which is 0.178 m (see Section 4.1). This results in a ratio that is used as the scaling factor from road model to environment when displayed as a floating point number. This factor is used to scale the x-axis of the environment, as seen in Table 4.6.

Table 4.6: X-axis scale factor

Parameter	**Value**	**Unit**
Width TB	0.178	m
Width Golf VII (incl. Mirror)	2.073	m
Width Scale Factor (X-Axis)	**0.086**	

The second factor is based on the ratio of the Golf's maximum speed and the TB's maximum speed. For this, it's pretended that the real car can drive with a maximum speed of 100 $^{km}/_{h}$. Using the real maximum speed of the Golf VII model would result in an impractical environment y-axis scaling. For example, the scaling would be so small, that every millimeter traveled in the environment would equal 0.333 m traveled in the real environment, making the driving functions hard to comprehend. Therefore, a custom maximum speed was established as the factor used to scale the y-axis of the environment as seen in Table 4.7.

Table 4.7: Y-axis scale factor

Parameter	**Value**	**Unit**
Pretend Speed	100.000	$^{km}/_{h}$
Pretend Speed	27.778	$^{m}/_{s}$
Real TB Speed	0.200	$^{m}/_{s}$
Speed Scale Factor (Y-Axis)	**0.007**	

In order to use the same factor on Speed and Length, the regular TB length is scaled down as seen in Table 4.8.

Table 4.8: Scaling the TB length

Parameter	**Value**	**Unit**
Actual Length TB	0.138	m
Length Golf	4.284	m
Length Scale Factor (same as Speed)	0.007	
Presumed Length TB ($ScaleFactor \cdot LengthGolf$)	**0.031**	m

The factors of the other strategies were also calculated for research purposes, but are not shown here for reasons of simplicity, since they were not settled upon as the used strategy.

4.2.2 Disadvantages of the Chosen Strategy

Five other scaling strategies were contemplated, see Section 4.2, but ultimately it has been settled to use the described „Width-Length-Scaling“ approach, because the disadvantages of it seemed manageable - more so than those of the other strategies. However, there are disadvantages to the chosen „Width-Length-Scaling Strategy“.

The selected „Width-Length-Scaling Strategy“ has its limitations and results in inconsistencies when applied to vehicle models that deviate from the dimensions of the Golf VII, such as the Jaguar. Comparing such different sized vehicles to the TB's dimensions leads to discrepancies in the resulting scaling ratios. However, the Golf VII is a good reference point for modeling the environment, since it is one of the most popular cars in Germany [42]. The error resulting from using other vehicle models in an environment that is scaled to the dimensions of the TB is expected to be neglectable. This is supported by the fact that only the y-axis scaling factor is derived from the ratio of the TB to a standard car model. That means, swapping in another vehicle configuration would only affect the scaled y-axis parameters. However, it's worth noting that, in the real world, vehicles of various sizes are commonly used, even though the typical German highway may be designed with a standard vehicle size in mind. Therefore, this issue is not addressed further in the ongoing work of this project group.

The „Width-Length-Scaling Strategy“ additionally results in a very short TB (3 cm), which is shorter than it is in reality (13.8 cm). However, this allows the usage of only two scaling factors for the three dimensions x-axis, y-axis, and speed, making further calculations easier. Using only two scaling factors is the main benefit of the Width-Speed-Scaling approach.

Since it is not possible to shorten the TB's dimensions to the resulting 3 cm in reality, this limitation is addressed by using the TB's actual length in critical components such as the ACC. Furthermore, e. g. , the MVM currently uses the scaled length of 3 cm, to make various calculations, such as the air resistance.

In summary, each analyzed scaling strategy presented its own set of disadvantages. The „Width-Length-Scaling Strategy“ was chosen because its disadvantages were deemed to be the most manageable.

If the described disadvantages prove to be very impacting on further advancements, there is another strategy which might be considered. One could use the „Three Factor Scaling Strategy“, but lengthen the TB to 0.37 $^{m}/_{with}$ a cardboard box wrapper. This way, only two scaling factors would be used, since the length scale factor would be the same value as the width scale factor, but the TB would resemble its emulated size also in the physical world. The cardboard boxes should be designed to remain stable during maneuvers, and should resemble the TB's design.

Table 4.9: Scaled environment parameters

	Actual	Axis	Scale Factor	**Scaled**	
Lane Width	3.750	X-Axis	0.086	0.322	m
Car Width	2.073	X-Axis	0.086	0.178	m
Roadway Width	11.250	X-Axis	0.086	0.966	m
Dashes Width	0.300	X-Axis	0.086	0.026	m
Speed	27.778	Y-Axis	0.007	0.200	$^{m}/_{s}$
Roadway Length	694.000	Y-Axis	0.007	4.997	m
Dashes Length	6.000	Y-Axis	0.007	0.043	m
Dashes Gap	12.000	Y-Axis	0.007	0.086	m
Car Length	4.284	Y-Axis	0.007	0.031	m

4.3 Scaling Reality

After defining the scaling strategy, and calculating the scaling factors in Section 4.2.1, in this section, the actual environment parameter values are calculated.

The table Table 4.9 shows the specifications of both the scaled and the original environment, with the respective scale factors from the „Width-Length-Scaling Strategy“ used. The lane width of a German highway is 3.750 m in reality, whereas it is 0.32 m in the scaled down environment, making the environment constructible.

4.3.1 Resulting Environment

In order to represent a real car in a smaller environment, the real models and Gazebo models of a three-lane highway use the dimensions depicted in figures Figure 4.1, which are based on the scaled environment parameters from table Table 4.9.

The resulting width of the scaled environment also fits a TB 3 Waffle model, since that model has a width of 30.6 cm. That means, the project group is not limited to using TB 3 Burger models only. E. g. , Waffles could be used as trucks in the environment.

Figure Figure 4.1 displays a graphical representation of the scaled environment using the calculated values from table Table 4.9

The environments are depicted as in Figure 4.2 and Figure 4.3.

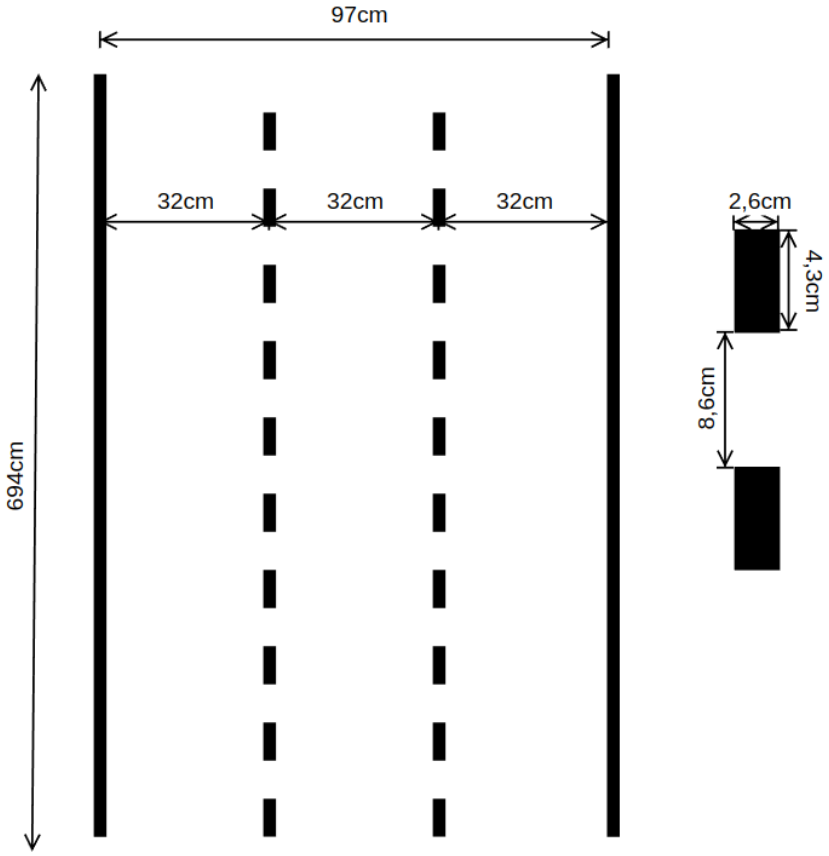


Figure 4.1: Specifications of the straight highway environment

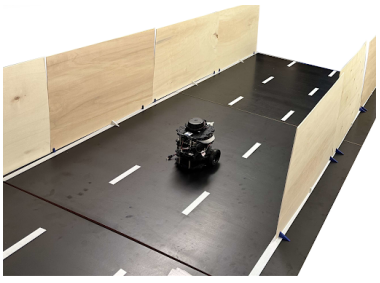


Figure 4.2: Road in reality

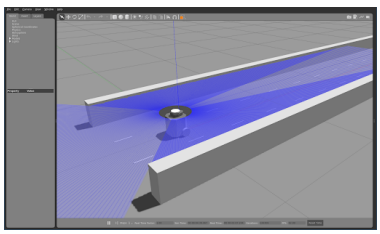


Figure 4.3: Road in the simulated environment (Gazebo)

4.3.2 Using the Scaling in the TurtleCar-Software

In opposition to Section 4.3, where only the scaling down of the road model using the Width-Speed-Scaling strategy is described, this section goes into detail on how the scaling strategy is implemented in TurtleCar.

To apply the scaling discussed in Section 4.2, the x and y values of all points must be multiplied by the scale factors. Since different factors scale the axes, it's necessary to adjust all angles as well.

A single scaling factor solution has been established quite early in the project group, as early as the preliminary project. However, research and experiments into the consequences on what happens, if two different factors for X and Y are actually used, have been made quite late in the project. Changing the scaling factors at such a late stage could have caused many unpredictable issues. As a result, the approach which uses a single scaling factor used for both axes has been kept. Specifically, the speed scale mentioned in Table 4.7 has been used as a single scaling factor.

This is a shortcoming of the project group. The mistake has been made to not include the researched scaling strategy earlier, as early as the research of the „Width-Speed-Scaling Strategy“. The problem was ignored, which resulted in many driving functions being developed on the state of a single scaling factor, such as the Model Predictive Control (MPC). This results in undefined problems which would occur when changing the single factor to two factors different for each axis.

This situation results in an environment, which is scaled using the Width-Speed-Scaling strategy, and the TurtleCar software, which uses the speed scaling factor only. No problems occur, since the environments scaling is not directly coupled to any specific technicality. The TurtleCar software just uses the environment as-is, without any specific reasoning. However, further efforts should aim at using the same scaling strategy for both creating the environment and scaling the coordinates in the TurtleCar software, since the current state is not consistent and origin of poor methods.

4.4 Differences between Gazebo and Reality

Various aspects of the Gazebo simulation lead to inevitable differences between it and the actual real-world setup. The following differences between the two have been identified:

- Gazebo has a continuous guardrail. In the real environment, this is approximated with smaller straight segments, which can lead to inaccuracies especially in curves. If the segments are placed too far apart, it can happen that a hole in the wall is detected, resulting in calculating wrong lanes.

- The small stands for the guardrail segments in the real environment currently reach into the lanes. This provides a potential hazard to the TB at the moment.

- The obstacle in Gazebo is 33 cm x 100 cm while in the real environment it is 32 cm x 44 cm. This means that the real environment obstacle doesn't completely fill a lane.

- The positions of lanes and lane markings in Gazebo are according to the measurements in Figure 4.1. In the real environment, they might be different depending on how precisely they are set up. The ground segments in real life are not perfectly flat, resulting in small inaccuracies in the lane widths.

4.5 Terms of Safety

The terms of safety for the vehicle follow those of the StVO [14].

4.5.1 Front Back Clearance

The distance to the vehicle in front or behind is not precisely defined by law. According to §4 Abs. 1 StVO, „The distance to a vehicle driving in front must generally be so large that it can be maintained behind it even if it suddenly brakes.“

However, there are some general rules of thumb for orientation: One rule suggests that the distance should be at least half the value shown on the speedometer, which is based on §2 Abs. 3a StVO. The second rule suggests that within urban areas, the distance should be 1 s, which at a speed of 50 $^{\mathrm{km}}/_{\mathrm{h}}$ is approximately 15 m. Outside urban areas, the distance should be 2 s, which at a speed of 100 $^{\mathrm{km}}/_{\mathrm{h}}$ is approximately 50 m.

4.5.2 Lateral Clearance

Regarding the lateral distance during overtaking, the legal guidelines are vague. According to §5 Abs. 4 StVO, „When overtaking, a sufficient lateral distance to other road users must be maintained.“ For overtaking „pedestrians, cyclists, and operators of small electric vehicles“ with motor vehicles, the guidelines are more specific: As per §5 Abs. 4 StVO, the sufficient lateral distance within urban areas must be at least 1.5 m, and outside urban areas, it must be at least 2 m.

4.5.3 Guidelines for the vehicle

In Table 4.10 are the clearance guidelines for the vehicle defined, which consider the regulations of the StVO. These are not complete (yet) and only are considering the scenarios which will be implemented.

Table 4.10: Clearances

Direction	**Clearance**	**Notes**
Longitudinal	$\frac{<\text{speed [km/h]}>}{2\ [\text{m}]}$	Rule of thumb: „half speedometer“ (§2 Abs. 3a StVO)
Lateral (overtaking car)	1 m	No concrete source could be found. There is a consensus among the internet on the value of 1 m.
Lateral (overtaking bicycle, in town)	1.5 m	§5 Abs. 4 StVO
Lateral (overtaking bicycle, out of town)	2 m	§5 Abs. 4 StVO

Chapter 5

Vehicle Emulation

In order to be able to develop controllers for real cars and test them on the TB, the behavior of a real vehicle needs to be emulated on the bot. This section describes the MVM used for emulation, its dynamics and its control inputs. First, an idealized MVM is defined, which is an abstraction of the actual emulation that is used for controller development. Afterward, the differences between the idealized model and the actual implementation are discussed.

5.1 Idealized Mathematical Vehicle Model

The vehicle emulation is based on the Kinematic Bicycle Model (KBM). The bicycle model is a simplified representation of a car's dynamics that is used in the field of vehicle dynamics and autonomous driving for motion planning and is also often used for model-predictive control [66]. It is called the „bicycle model" as it consolidates the dynamics of a car into a two-wheeled model, where the two front wheels and the two rear wheels are each represented as a single wheel. The model can be defined from different points of view along the vehicle. For this project group, a model with a viewpoint from the center of the rear axle was chosen for reasons of simplicity. For controlling the lateral movement, the bicycle model uses steering angle as input. The model used here is based on the bicycle model definition given by Polack et al. [66]. It contains the following variables:

- X and Y: The longitudinal and lateral positions of the vehicle, respectively
- θ: The heading angle of the vehicle
- v: The speed of the vehicle
- a: The acceleration of the vehicle
- α: The steering angle
- l: The wheelbase length, which is the distance between the front and rear axles
- r_1: Tire friction value

- r_2: Air resistance value
- d_1: Linear approximation of the disturbance on the acceleration due to motor friction and transmission

The variables l, α, r_1 and r_2 are all parts of the vehicle configuration. For model simplicity, the acceleration in this model is only affected by friction forces. Motor resistance is only considered as a linear disturbance parameter d_1 acting on the acceleration input in this idealized model. These errors were chosen to only affect acceleration, not steering.

The control inputs to the model are defined as follows:

$$u_1 = a$$
$$u_2 = \alpha$$

The dynamics are defined as follows:

$$\dot{x_1} = \dot{X} = x_3 \cdot \cos(x_4)$$
$$\dot{x_2} = \dot{Y} = x_3 \cdot \sin(x_4)$$
$$\dot{x_3} = \dot{v} = r_1 \cdot x_3 + r_2 \cdot x_3^2 + d_1 \cdot u_1$$
$$\dot{x_4} = \dot{\theta} = \frac{v}{l} \cdot \tan(u_2)$$

In the beginning of the project group, a simplified model with three instead of four states had been used. The velocity had been modeled as a control input and the implementation was based on using always the maximum acceleration and deceleration to achieve that velocity as quickly as possible. Because this only allowed for very sharp and inconvenient driving maneuvers, the model was reworked to use the acceleration as an input, as it gives the controllers more freedom and represents the input of a real car more closely.

5.2 Emulated Mathematical Vehicle Model

The emulation extends the idealized model described in Section 5.1 in order to make it more realistic. In addition to friction and wind resistance, it contains an approximation of a car's transmission and gear shift, which affects the actual acceleration of the car. These disturbances are not part of the idealized model. Therefore, controllers need to be designed in a way to be robust against these errors and model discrepancies.

In order to provide these realistic disturbances, a motor and friction model is used to calculate the highest acceleration of the vehicle possible given the current gear, vehicle weight, velocity and other relevant variables. If the acceleration given by the controller is higher, the maximum possible acceleration of the vehicle is used instead. Conversely, if the vehicle is braking (the controller is giving a negative acceleration), the highest possible deceleration is calculated and the minimum acceleration is used.

The model and its implementation are described in the following.

5.2.1 Motor Model

The motor model is implemented through a series of calculations and functions which account for various factors including the engine's revolutions per minute (RPM), torque and gear ratios. The motor model takes into account car-specific factors which can be configured to emulate different types of cars.

The following calculations are based on the formulas given by Stark [84].

Engine RPM The engine's RPM is computed based on the vehicle's current speed, the wheel circumference, and the current gear and final drive ratios.

1. Convert the speed from meters per second to meters per minute by multiplying with 60:
$$v[\mathrm{m/min}] = v[\mathrm{m/s}] \cdot 60$$
2. Calculate the wheel's revolutions per minute (RPM) P_{wheel} by dividing the speed by the wheel circumference C:
$$P_{wheel}[\mathrm{rpm}] = \frac{v[\mathrm{m/min}]}{C[\mathrm{m}]}$$
3. Finally, calculate the engine's RPM P_{engine} by multiplying the wheel's RPM with the current gear ratio G and the final drive ratio D:
$$P_{engine}[\mathrm{rpm}] = P_{wheel}[\mathrm{rpm}] \cdot G[\mathrm{rpm}] \cdot D[\mathrm{rpm}]$$

Torque The current torque is calculated based on the engine's RPM. A linear interpolation function is employed to interpolate the torque values from a predefined set of engine speed and torque points.

Gear Shift Handling The motor model checks whether a gear shift is available or necessary based on the current RPM and the specified RPM ranges for each gear. If a gear shift is required, the current gear is updated, and the time of the last gear switch is recorded.

Engine Acceleration Force The engine acceleration force is the total force provided by the engine and is calculated using the current torque, gear ratio, final drive ratio, and the wheel radius. This calculation accounts for transmission losses.

1. Compute the engine torque after transmission $T_{engine,t}$ by multiplying the average engine torque $T_{engine,avg}$ with the gear ratio G and the final drive ratio D:
$$\begin{aligned} T_{engine,t}[\mathrm{Nm}] = T_{engine,avg}[\mathrm{Nm}] \\ \cdot G[\mathrm{rpm}] \cdot D[\mathrm{rpm}] \end{aligned}$$
2. Compute the engine torque after accounting for engine losses $T_{engine,l}$ by multiplying the engine torque after transmission with the engine loss factor LF_{engine}:
$$T_{engine,l}[\mathrm{Nm}] = T_{engine,t}[\mathrm{Nm}] \cdot LF_{engine}$$

3. Finally, calculate the engine acceleration force $F_{a,engine}$ by dividing the engine torque after losses by the wheel R R_{wheel}:

$$F_{a,engine} = \frac{T_{engine,l}[\text{Nm}]}{R_{wheel}[\text{m}]}$$

The TB's linear and angular velocities can be controlled to emulate the motion of a vehicle as described by the Bicycle Model. The Bicycle Model is used since the TurleBot only has two wheels. The model's state variables are mapped to TB controls as follows:

- The longitudinal velocity v of the model corresponds to the linear velocity of the TB.
- The heading θ of the model is used to control the angular velocity of the TB

5.2.2 Steering Angle Limiting

To ensure that the simulated vehicle behaves realistically at high speeds, its steering angle needs to be constrained depending on its driven speed. The steering angle directly affects the turn rate of the vehicle; a high steering angle combined with high speed results in high lateral acceleration and consequently could lead to a loss of control over the vehicle. To avoid this and ensure that the vehicle's behavior is predictable, the allowed lateral acceleration of the vehicle needs to be restricted in the emulation model.

BOSETTI ET AL. present a formula for the accepted lateral acceleration as a function of vehicle speed [10]. The formula uses a criterion taken from LEVISON ET AL. using data from a driving behavior study to model the acceptable lateral acceleration for an average and a 85^{th} percentile driver ($K = 42.0$) [48]. The lateral acceleration limit imposed by the criterion can be seen in Figure 5.1. The term „85th percentile driver“ here refers to someone driving more dynamically than 85% of the population. This solution is suitable for setting a velocity-based limit on the steering angle in the used MVM, as it is slip free and relatively simple. For more complex vehicle models, additional factors such as tire dynamics and road conditions could be considered in order to limit the steering angle in a way that ensures safety. This alternative approach would be independent of the drivers comfort and instead based on vehicle configuration and road conditions.

The calculation of the maximum acceptable steering angle in the model is as follows:

1. **Radius**: The turning radius R of the vehicle given a steering angle α and the vehicle's wheelbase l is calculated as

$$R = \frac{l}{\tan(\alpha)}$$

The turning radius is inversely proportional to the tangent of the steering angle, which follows from the Bicycle Model.

2. **Current Lateral Acceleration**: Given the vehicle's velocity v and turning radius R, the lateral acceleration a_{lat} experienced by the vehicle can be calculated using the centripetal acceleration formula:

$$a_{lat} = \frac{v^2}{R}$$

3. **Accepted Lateral Acceleration**: The accepted lateral acceleration for a given velocity is calculated using the criterion K from Levison et al.. For an average driver, the K criterions has been estimated to be 36.0 and for the 85$^{\text{th}}$ percentile driver 42.0. The formula from Levison et al. is as follows:

$$a_{lat\_accepted} = \left(\frac{K}{v}\right)^2$$

If the lateral acceleration for a given speed remains below this value, that means the vehicle's behavior is within safe limits [48].

4. **Steering Angle**: If the current lateral acceleration exceeds the accepted value, the steering angle α must be reduced. This is done by first calculating the maximum allowed turning radius R_{max} using the accepted lateral acceleration (formula derived from step 2):

$$R_{max} = \frac{v^2}{a_{lat\_accepted}}$$

Subsequently, the required steering angle to achieve this turning radius is found using:

$$\alpha_{max} = \arctan\left(\frac{l}{R_{max}}\right)$$

which gives the maximum permissible steering angle.

Additionally, the model incorporates a static maximum lateral acceleration value of $5\,\mathrm{m/s^2}$. This limit was derived based on the Modified Levinson's criterion visualized by the orange line in the graph in Figure 5.1 and was imposed to improve the low speed behavior of the simulated vehicle.

5.3 Vehicle Configuration

Vehicle configurations, which contain all parameters necessary to simulate a realistic vehicle, are used. These can be switched out depending on the simulated scenario. Each configuration file is written in the YAML language and contains the parameters for a specific vehicle model. Furthermore, the configuration of each vehicle includes details regarding individual components, such as the specific engine model. This modularity enables constructing configurations using various types of vehicle parts that have already been defined. Currently, two different configurations are used, one for simulating a sports car (Jaguar F-Type) and one for a more casual car (VW Golf VII). This way, the implemented driving functions can be tested on a range of car types: fast and slow ones. In this chapter, the structure of a vehicle configuration is described and the two used configurations presented.

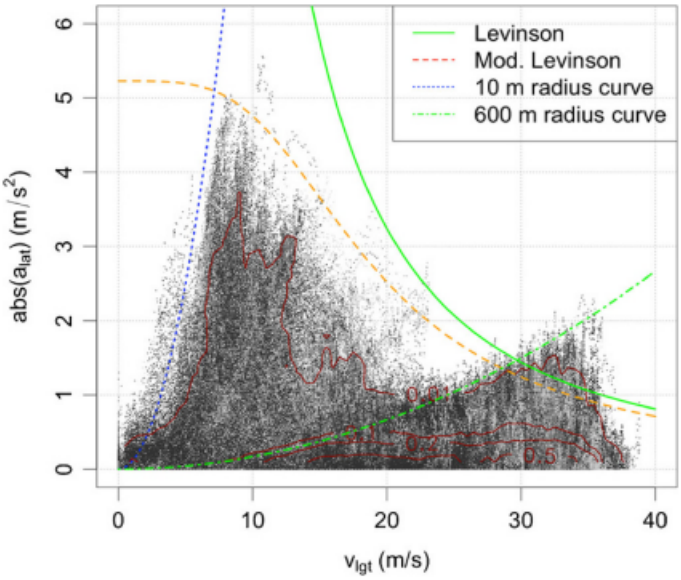


Figure 5.1: Lateral acceleration in relation to speed based on study data [10]. The green curve is the limit calculated using Levinson's criterion for an average driver. The orange curve shows the Modified Levinson's criterion [48].

5.4 Structure

The modules describing vehicle parameters are divided into five categories: engine, transmission, wheels and vehicle dynamics. These are described in the following.

Engine The engine is a component that is used by all vehicles and usually varies from vehicle to vehicle, therefore, its parameters need to be specifiably independently. The engine specification can be found in Table 5.1.

Table 5.1: Engine Specifications

Parameter	**Unit**
Max torque	Nm
Speed at maximum torque	rpm
Maximum power	hp
Speed at maximum power	rpm
Average torque	Nm
Average loss	%
Speed points full load	rpm
Static torque points full load	Nm

Transmission Similar to the engine, transmissions are usually unique across different vehicle models. The transmission specification can be found in Ta-

ble 5.2.

Table 5.2: Transmission Specifications

Parameter	**Unit**
Gear switch time	s
Start speed	rpm
End speed	rpm
Gear Ratio	Multiplier value
Final drive ratio	Multiplier value

Wheels The wheels are a highly variable component when comparing different vehicles. The wheel characteristics can be found in Table 5.3.

Table 5.3: Wheel Characteristics

Parameter	**Unit**
Wheel radius	m
Wheel circumference	m
Friction	N

Vehicle Dynamics The remaining parameters are dependent on the whole car. They can be found in Table 5.4.

Table 5.4: Vehicle Dynamics and Performance

Parameter	**Unit**
Maximum steering angle	rad
Wheelbase	m
Width with mirrors	m
Braking force	N
Mass	kg
Air resistance	N
Aerodynamic drag	N
Frontal area	m^2
Maximum speed	$^{km}/_{h}$
Acceleration time 0 to 100 $^{km}/_{h}$	s

5.5 Examples

The two configurations already implemented and used in this project are described in the following.

VW Golf VII The Volkswagen Golf MK7 with a 2.0 litre diesel engine was selected to represent a casual everyday car compared to the rather sporty Jaguar F-Type. Some specifications are depicted in Table 5.5. They correspond to models built from 12/2016 to 05/2020, and mainly influence the motor and transmission type. There has been a facelift in 2017, which slightly changed the exterior und integer design, but has no impact on technical parameters. Some of those specifications are not strictly bound to the car model itself, i. e. the transmission „DQ381“ is used in many other vehicles.

Table 5.5: VW Golf VII Model Details

Parameter	**Details**
Model	VW Golf VII 2.0 TDI with DSG
Build Duration	12/2016 - 05/2020
Engine Type	Diesel
Engine Series	VW EA288
Engine Code Letters	CRMB, DCYA, DEJA, CRLB
Displacement	1968 cm^3
Max. HP @ RPM	$150 @ 3500 - 4000$
Max. Torque @ RPM	$340 @ 1750 - 3000$
Used Wheel Size	205/55 R16
Transmission Type	DQ381
Remarks on Transmission	DSG with 7 gears (From 12/2026, 6 gears previously)
Drive Type	Front wheel drive

Jaguar F-Type The configuration of this vehicle model originates from the pre-project and contains the specifications of a Jaguar F-Type. These specification are depicted in Table 5.6. The Jaguar F-Type was selected to represent a sports car.

Table 5.6: Jaguar F-Type Specifications

Parameter	**Specification**
Model	Jaguar F-Type
Engine type	3-litre V6 DOHC V6
Max. HP @ RPM	340 @ 6500
Max. torque @ RPM	450 @ 3500
Used wheel size	295/30 R20
Transmission type	Automatic, ZF8HP, RWD
Drive type	Rear-wheel drive

The information for the Jaguar F-Type is taken from X-ENGINEER [94]. However, the final drive ratio parameter provided by this source is not used. It specifies a single value for all gears. The VW Golf, on the other hand, has two different values for that parameter instead of a single global one for each gear. Due to this fact, the Jaguar F-Type configuration was adapted to represent this structure by copying its own value.

Clear information on some parameters for the VW Golf could not be found. These were taken from general knowledge or approximated as described in the following. The steering angle is defined by a statement in Auto Motor und Sport: „40 degree steering angle for conventional vehicles“ [6]. The braking force is calculated by taking an intermediate value for the deceleration between the emergency braking value for the Golf, which is about 10.6 $\mathrm{m/s^2}$ [13], and the minimum required value by law: 2.5 $\mathrm{m/s^2}$ [12].

Also, the values for the VW Golf's parameters had to be gathered by different sources. These will be mentioned here. The aerodynamic drag, frontal area and air resistance values are taken from the collection of Cordes [17]. The gear switch time values are taken from VWVortex [22]. Additional sources were used for information about the gear ratios [21], [79]. The dyno chart was taken from More BHP. It shows two graphs, the important one is the thick line representing the stock engine [7]. The chart was manually evaluated using Engauge Digitizer [53]. Information on wheels and further details were also collected [92], [87].

5.6 Considered Mathematical Vehicle Models

In this section, other applicable mathematical vehicle models MVM are discussed and compared to the used KBM KBM based on extensive research in the field of vehicle dynamics. At the end of this section, one should have an overview of different vehicle models, their advantages and disadvantages, and how applicable they are to this project. Also, ideas for further evaluations and experiments are given, which could lead to improvements of the TurtleCar platform.

The research on this topic has been extensively guided by the following main questions.

1. Are other MVM more fitting to the task at hand?
2. Should the Differential Drive Model (DDM) be used instead?
3. What are the consequences of using a simple KBM, and not, e. g. , a complex four-wheel dynamic model, and, are these consequences acceptable?
4. Which replacements can be made to address certain inaccuracies, like missing tire slip and mechanic forces?
5. Which further evaluations and experiments could be made to test the effectiveness of the used model?

5.6.1 Kinematic Bicycle Model in the Field of Mathematical Vehicle Models

The KBM is a kinematic MVM, in opposition to dynamic MVM. Kinematic models describe the geometrical change over time in a planar system like an inertial coordinate system. Dynamic models on the other hand describe the interaction of forces and mass distributions which result in velocity and acceleration. The KBM is commonly used for creating controllers for model-predictive control [66].

The KBM is a MVM with low complexity. It has few Degrees of Freedom (DOF), in general two to six. Some MVM exist with up to 19 DOF [76, p. 313], so in comparison, the KBM is a simple model. It does not incorporate roll or pitch angles, and only considers the frontal wheel as a steering wheel.

Literature suggests, that for low complexity driving functions, the KBM should be sufficient: It can be used to represent four-wheeled cars as well as two wheeled cars with almost equal accuracy. The number of DOF for the KBM does not seem to play a big role in regular applications. A complex six DOF Bicycle Model is very close in results to a simple two DOF model. The KBM is also flexible. It can be used to model longer wheelbase vehicles such as buses. Other papers have shown that a four-wheel steering double track model, as an extension of the Bicycle Model, does result in higher accuracy in some situations. However, these advantages are only relevant for inexperienced drivers. Furthermore, front wheel steering provides most of the benefits of four-wheel steering [24, p.215].

However, having stated benefits of the KBM, more complex models are able to represent reality more closely, which might lead to higher accuracy and more capabilities. Siwek et al. support this argument, proposing that dynamic models are much better solutions than kinematic models [80].

5.6.2 Reasons for Choosing the Kinematic Bicycle Model

First of all, it should be noted that the KBM has been chosen for simplification reasons. The assumption has been made that it is a model which should suffice for all kinematic descriptions of motion in the context of this project group. However, critical reflection is required because the chosen MVM has a big impact on the TurtleCar software and its set of all implementable driving functions.

Even though the goal of this project group is to simulate real world cars using the inexpensive hardware of the TB, deviations will naturally occur, since the TB only has one wheel axis and does not conform to the regular geometry of a car. This means, simulation of driving functions using TurtleCar inherently does not adhere to higher goals of perfect simulation, but rather to practical and economic reasons.

Therefore, whether a very complex or highly simple MVM is chosen does not really matter for the goals of TurtleCar. However, a simple MVM most likely results in simpler implementations of driving functions and reduces complexity, the presence of which could rise the entry hurdle of future developers and students. This advantage of lower complexity is the main reason why the KBM has been chosen.

Moreover, the KBM presents itself as a natural choice, because most robots have low level controllers built into them, which hold a steady velocity once given [20] - just like the TB does. Since the KBM proposes differential equations for the inputs of speed and acceleration, but most dynamic models define the inputs to be torques and voltages [54], choosing a dynamic model imposes even more difficulties.

On the other hand, an insufficiently complex MVM could be the cause of driving functions not working as intended or not being able to implement future features. Stating common replacements for the KBM and their possible use in TurtleCar could support further work on the TurtleCar platform and give an idea for its possible improvements. If one chooses to include a dynamic model,

there appears to be research on providing speed and acceleration input based dynamic models [60].

5.6.3 Discussion of other Vehicle Models

In this subsection, other MVM are discussed as alternatives to the KBM. First, other possible options are introduced. Second, arising questions are discussed.

Differential Drive Model This is a model which aims to predict the behaviour of vehicles with a differential drive. A differential drive is one where the vehicle has only one axle and optionally a ball caster wheel for stabilization [54]. This is the case for the TB as well. This model would be a good fit for steering a differential drive robot. However, the focus of the project group are four-wheeled cars without a differential drive. Using a DDM would therefore not be a suitable basis, since it would require a translation layer to a non-differential car model.

Dynamic Bicycle Model This model is the dynamic version of the KBM, considering not only geometric change, but also torques and forces. Kang et al. provide an in depth comparison of the properties of both and their results. They state that for autonomous driving in highway conditions, „unless there is considerable variation in tire-road friction, the kinematic model is enough to understand and analysis of vehicle motion“ [38].

Four-Wheel Kinematic Model This model is an extension of the KBM, in considering not only two wheels, but four. This might result in better representation of real cars. However, the achieved benefit of four-wheel steering is only relevant in certain conditions [46]. Also, four-wheel steering benefits originate for the greatest part from the front wheels [1].

Four-Wheel Dynamic Model This model is the dynamic version of the Four-Wheel Kinematic Model and thus provides the same benefits over the kinematic model as the Dynamic Bicycle Model would on the KBM. If a dynamic model should be chosen, and a four-wheel model is required, this would be a good choice. As stated above, using a dynamic model is not important to TurtleCar, since the application does not incorporate heavy and fast vehicles. The TB weighs 1 kg and its maximum translational velocity is $0.22\,{}^{\mathrm{m}}/_{\mathrm{s}}$, which cannot be compared to real cars for which Kong et al. [41] make their arguments. Additionally, TB has almost no side-slip angle due to its physical proportions, so a dynamic model would not yield extra accuracy.

5.6.4 Possible Reasons for Choosing other Models than the Kinematic Bicycle Model

Martins and Brandão [60] suggest, that high velocity or heavy load transportation applications should use dynamic models. Applications that contain inexperienced drivers should consider four-wheel models, as suggested by Eskandarian [24, p.215]. Siwek et al. [80] advocate dynamic robot models for high velocity and high positional accuracy applications. Kong et al. [41] on the other hand

recommend the Dynamic Bicycle Model for controllers that produce insufficient results or are confronted with higher speed sinusoidal tracks. When faced with high lateral acceleration scenarios, Polack et al. [67] propose to use higher accuracy models. Dynamic models should also be chosen when forces like torque and tire side-slip angles need to be considered, as these are regarded in the dynamic counterparts of the KBM. On a different note, Martins et al. [54] state that choosing a dynamic vehicle model results in the ability to perform stability analysis on the vehicle, which might result in high-end, i. e. , very accurate, controllers.

As stated, the „Dynamic Bicycle Model“, „Four-Wheel Kinematic Model“, and „Four-Wheel Dynamic Model“ would not yield significatn benefits for the scope of this project. However, whether using the DDM or the KBM is more complex. First, it is a good thing that the TB is using a differential drive, as they can perform more maneuvers and are more flexible in tight spaces [59]. One could then ask why a DDM should not be used as well. An argument against that would be that the project group aims to provide a platform for experiencing the development of real-car controllers, which are used in four-wheel vehicles. So, the MVM that is used as a basis should also be close to those vehicles, for which, as stated in Subsection 5.6.1, the KBM is sufficient. However, this argument requires more evaluation and experiments, since it has not been tested experimentally.

In conclusion, future developments and works on TurtleCar should consider the given suggestions when they are challenged with one of these problems.

5.6.5 Answering the Research Questions

In this section, the main research questions asked in the beginning of this chapter are answered based on the research in the sections above. For answering those, the context of the questions should be stated again. This project group is aiming to provide a platform for getting hands-on experience in the development of controllers for real cars. It is not aiming to develop optimal controllers and driving functions, but to provide a framework for developing interesting driving functions on inexpensive hardware and simulating real life driving scenarios in a controlled environment. Therefore, the achievable learning effect is valued higher.

Are other MVM more fitting to the task at hand? This depends on the emulated scenarios and driving functions to develop. For the current scenarios and driving functions, the KBM is sufficient. The implemented driving functions and controllers in this project group work well and there were no hints that the KBM imposes a problem. This is also backed by research [41]. However, there are scenarios where the used model should be replaced, as stated in Subsection 5.6.4.

Should the DDM be used instead? For this, more extensive experiments and research should be conducted in the future. For now, the KBM seems to model the TB sufficiently, since no problems were encountered. Also, the KBM is more close to the vehicles that are the aim of emulation, as described in Chapter 5, which should result in more transferable learning experiences in developing controllers for real cars. However, a more accurate way of controlling the TB itself would probably be using the DDM. It could be, that those benefits would only present themselves with bigger robots and higher speeds.

What are the consequences of using a simple KBM, and not, e. g. , a complex four-wheel dynamic model, and, are these consequences acceptable? The consequences have been described extensively in the sections above. They are acceptable, since the KBM seems to be sufficient for the developed driving functions in this project group. When issues occur, one could improve the model by following Subsection 5.6.4.

Which further evaluations and experiments could be made to test the effectiveness of the used model? A summary is given in Chapter 20.

Which replacements can be made to address certain inaccuracies, like missing tire slip and mechanic forces? This is answered extensively in Subsection 5.6.4. However, it should be mentioned that most improvements of driving functions would be made in tuning of the parameters and the controllers itself [78].

Chapter 6

Sensor Augmentations

This section describes various sensor augmentations made during the project group. The augmentations are ranging from hardware modifications to additional ROS topics.

6.1 Camera

The camera of the vehicle streams the image data in front of it into the ROS network. The camera data can then be used to perform lane and object detection in the frames sent by the camera. Lane boundaries and road participants are examples of objects to be detected.

6.1.1 Camera Service

The camera service is used to stream image data into the ROS network. It can be used in two ways:

- With a web server
- Headless

With a web server This variant uses a web server to show the image stream sent by the vehicle's camera on a webpage. The server is hosted on the TB's network address and is running on port 5000. To see the images, the webpage has to be refreshed once after starting the camera service. This variant is more suitable for troubleshooting.

Headless The second variant needs no user interaction for sending messages. If the ROS service `/camera_serice` is called it will either start or stop sending images.

Parameters In the service request, different values are used for parameters as „Opcodes“ to customize how the node should behave. These are listed in Table 6.1, Table 6.2, Table 6.3, Table 6.4:

Table 6.1: Parameters for the camera service request

Parameter	**Description**	**Default Value**
request	The request opcode	0
frame_width	The requested frame width of images	640
frame_height	The requested frame height of images	480
frame_rate	The framerate to capture images with	10

Table 6.2: Table for Opcodes

Opcode	**Description**
0	Toggels camera server

Table 6.3: Parameters for the camera service response

Parameter	**Description**	**Default value**
status	The response Code	0
message	A message with status information	N. A.

Table 6.4: Table for status codes

Opcode	**Description**
0	Success

6.1.2 Camera Mount

The TB already had a static mount for the camera attached. To be able to dynamically change the viewport of the camera, the simple mount was replaced by a more advanced mount. This new mount uses a pan-tilt design to make the camera angle adjustable on two axles. The new mount was designed for and printed using a 3D-printer. For the assembly, small screws were used and the servos were put in place. The 3D model can be seen in Figure 6.1. Since this new mount effectively, slightly reduced the viewport stability of the camera and the project group has not encountered a situation where an adjustable camera would be crucial, the basic mount still remained the overall default.

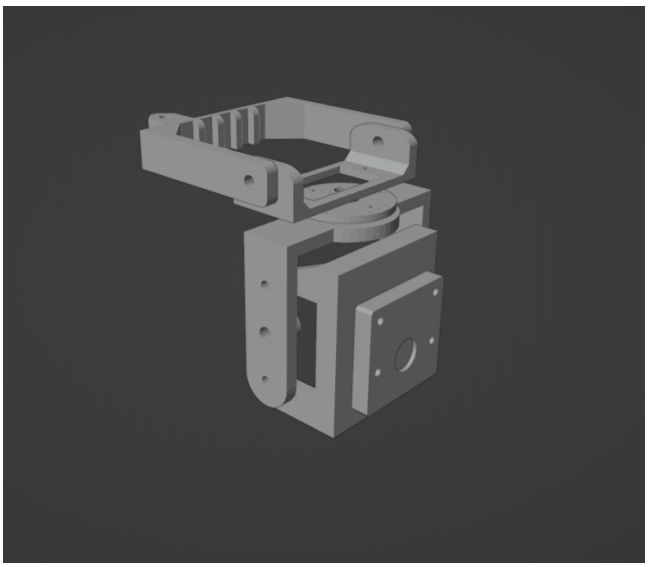


Figure 6.1: New camera mount with movable joints to control the camera with servo motors.

6.2 Kalman Filter

The TB uses the odometry topic `/odom` to publish information about the vehicle's position and movement. This data provided is however quite noisy. To counteract this problem, a state estimator was chosen to filter the noise and approximate the real state of the vehicle. A common approach for estimating the state of a moveable vehicle is using a Kalman Filter (KF).

The KF is a recursive algorithm designed for estimating the state of a linear dynamic system from a series of noisy measurements. At its core, the KF operates in two fundamental steps: prediction and update. These steps are mathematically represented by a set of equations that govern the filter's operation, allowing it to predict the system's future state and then correct this prediction based on new measurements [37].

A KF assumes a linear evolution of its state and expects a Gaussian distribution of error. The bicycle model, which is used to describe the movement of the vehicle, is a non-linear model. This means a KF would not be a viable use for the estimation of the vehicle's state.

For non-linear models there are three other options to use: Extended Kalman Filter (EKF), Unscented Kalman Filter, and Particle Filter. The EKF is chosen for its lower computational requirements compared to the alternatives.

The implementation of the EKF was based on the project „Kalman and Bayesian Filters in Python“ [45], which provided documentation on the creation of an EKF for tracking the movement of a robot.

The variables used for the vehicle state are:

- X and Y: Longitudinal and lateral positions of the vehicle, respectively
- θ: Heading angle of the vehicle

- v: Speed of the vehicle
- a: Acceleration of the vehicle
- l: Wheelbase length, which is the distance between the front and rear axles
- $\dot{\theta}$: Turn speed of the vehicle
- a_u: Acceleration control signal of the vehicle
- α_u: Steering angle control signal of the vehicle
- t: Time Step

The state and control matrix for the EKF are defined as followed:

$$x = \begin{bmatrix} X & Y & v & \theta & a & \dot{\theta} \end{bmatrix}^T$$
$$u = \begin{bmatrix} a_u & \alpha_u \end{bmatrix}^T$$

The predicted movement of the vehicle is described as followed:

$$f(x,u) = \begin{bmatrix} \cos\theta \cdot v \cdot t + x \\ \sin\theta \cdot v \cdot t + y \\ a \cdot t + v \\ \dot{\theta} \cdot t + \theta \\ a_u \\ v \cdot \tan\alpha_u / l \end{bmatrix} \tag{6.1}$$

The equations for the predicted movement are based on the bicycle model described in section 5.1.

In the update phase the EKF gets the velocity and heading of the vehicle provided by the odometer. The variance of the measured velocity is not constant and is depended on the actual velocity of the vehicle. Variance values were determined by measuring 100 data points for each velocity from $0\,\mathrm{m/s}$ to $0.2\,\mathrm{m/s}$.

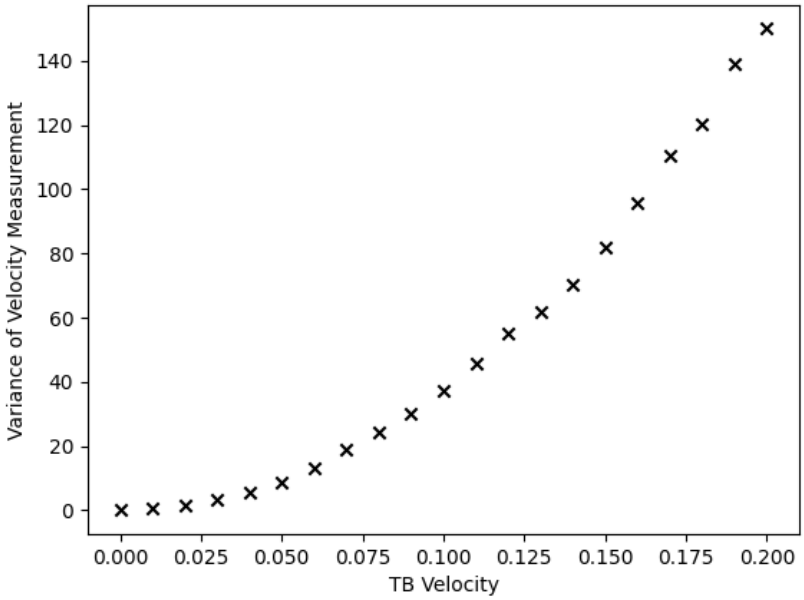


Figure 6.2: Variance of Velocity Measurement

Through graphical analysis it was determined that the variance function is quadratic. A exponential regression was done and resulted in the function:

$$var_v(v) = 3800 \cdot v^2 + 23.1 \cdot v + 0.4 \tag{6.2}$$

It was tried to incorporate the acceleration from the inertial measurement unit (IMU), but this effort was abandoned in favor of other important tasks. The IMU signals have strong outliers which occur because of the limited discrete velocities the TB can reach. The TB can reach speeds from $0\,\mathrm{m/s}$ to $0.22\,\mathrm{m/s}$ and can only increase or decrease its velocity in $0.01\,\mathrm{m/s}$ steps. This has the effect that the TB can only achieve 23 discrete velocities.[1] An acceleration of $0.01\,\mathrm{m/s^2}$ would result in a velocity graph like this:

[1] The effective velocity the TB reaches is not truly discrete due to environmental influences, but it can only be targeted as such.

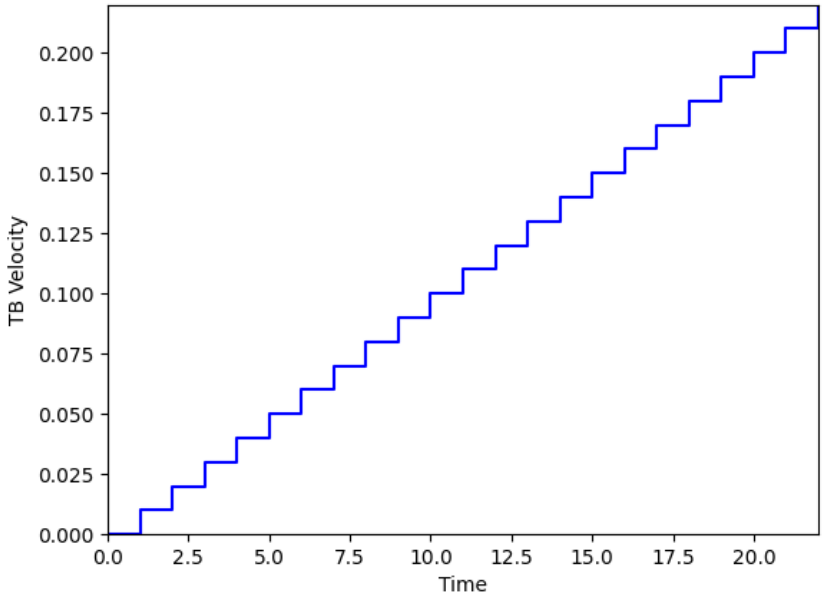


Figure 6.3: Velocity of TB

And the corresponding acceleration of the TB would look like this:

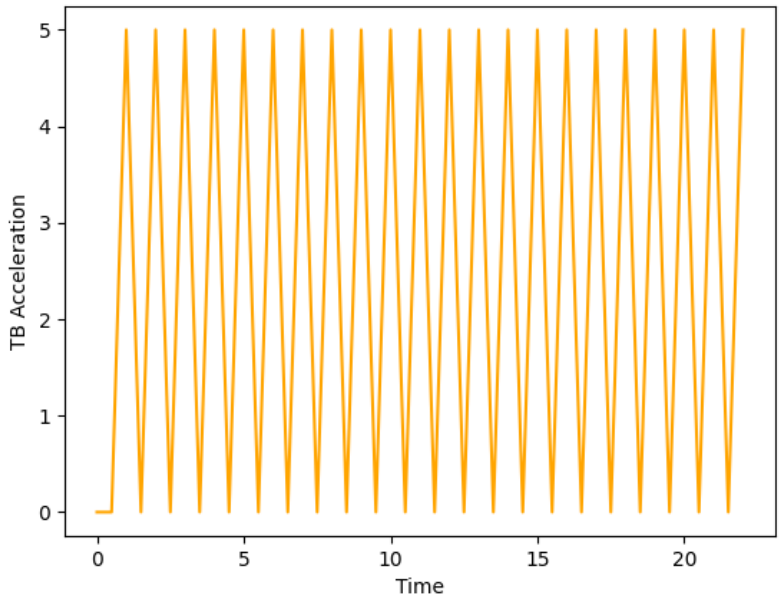


Figure 6.4: Measured Acceleration of TB

To measure the applied acceleration, the frequency of the peaks needs to be calculated. This is not an easy task if the measurement has a lot of noise and the acceleration keeps changing.

In figure 6.5 the resulting EKF in comparison to the measurements is seen.

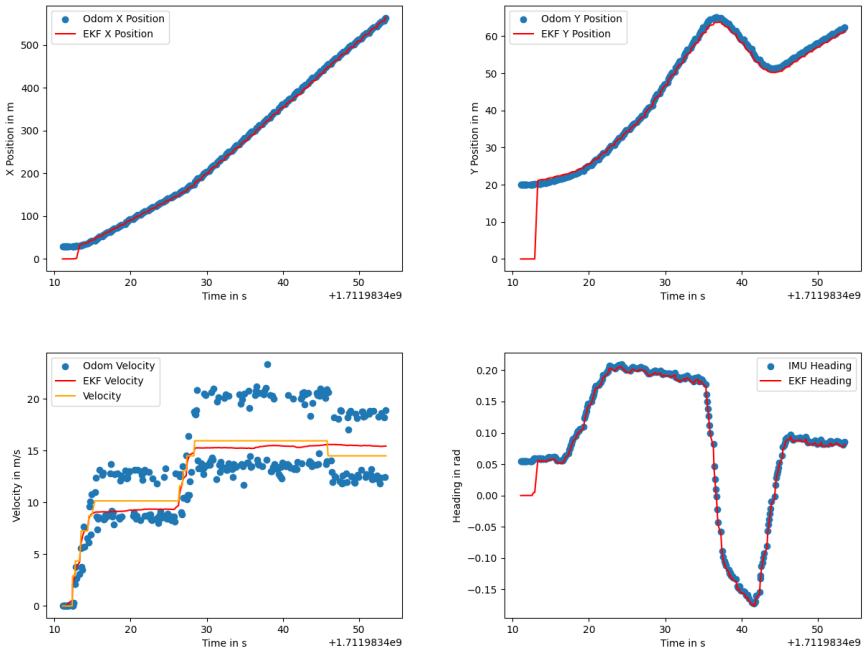


Figure 6.5: EKF: Vehicle State

The EKF provides a good estimation for the filtered states, especially for the velocity which has the highest variance in the measurement. The velocity mentioned inside Figure 6.5 is the velocity signal send to the vehicle by the transposer.

6.3 Sensor Fusion

Sensor Fusion is the combination of different sensor signals into one signal to create information which is more reliable than that provided by the sensors individually. There are several methods that can be categorized into three groups: „Probabilistic methods“, „Methods based on least squares techniques“, and „Fuzzy logic and neural networks“. The most common method is Kalman filtering which is part of the category „Methods based on least squares techniques“ [75].

In the following are some approaches described to introduce sensor fusion into TurtleCar-Core.

The TB uses four kinds of sensors to collect data about its environment and its own state: an odometer, an IMU, a LIDAR, and a camera.

Odometer and IMU provide data about the state of the vehicle, such as velocity, acceleration, X Position, and Y Position. This information is already

incorporated into an EKF, which is described in 6.2. With a KF, it is easy to fuse sensor inputs as it just requires multiple updates with the different sensor data.

The LIDAR and the camera collect information about the environment. There are two possible fusions for LIDAR and camera data: Fusing the lane information and fusing the obstacle information.

To fuse the lane information, a simple approach would be to take the measured lane points from the LIDAR and the camera and interpolate a B-spline through all points. Another approach would again be to incorporate a KF and update the filter with both camera and LIDAR data. The usage of a KF for lane boundaries is described in multiple papers [50] [35].

Obstacle information is already partially fused by combining some information from the LIDAR detection with the camera detection into one list of all detected obstacles. As described at the end of section 10.1, here the usage of a KF would be a viable approach.

Chapter 7

TurtleCar-Core

There are multiple ROS Nodes running on the TB which together form TurtleCar-Core. The architecture of each node is described here. In order to run these nodes, a specific setup of the image running on the TB is required, which is explained here as well.

7.1 TurtleCar Node

In the module called `TurtleCar-Core`, the main parts of the software controlling the TB are implemented. Its tasks are to gather sensor data, define a control action according to its current scenario and goal, and publish that action to the relevant actuators.

7.1.1 Architecture

The diagram in Figure 7.1 shows the basic building blocks of the code. It is simplified in the way that the *TurtleCarNode* class is the root class and consists of all other classes. In order not to clutter the diagram, these compositions are not drawn.

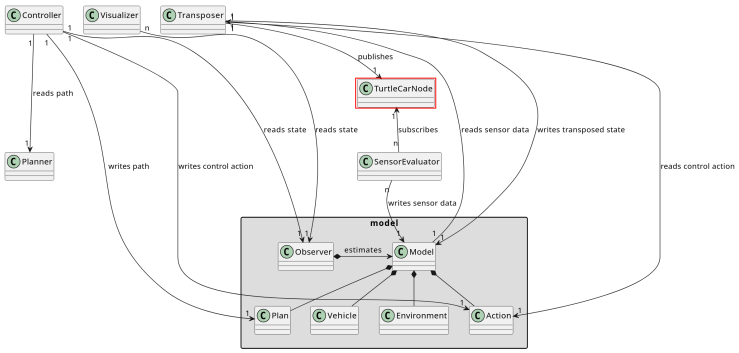


Figure 7.1: Static view of the architecture of the *TurtleCarNode*

The architecture is modular and can be separated into these classes:

- **TurtleCarNode:** The *TurtleCarNode* class is the root class. It provides the ROS interface which is used by other parts of the software to subscribe or publish to ROS topics. It is also the root for the tree of dependent classes.
- **Model:** The model represents the observable state of the robot. It contains information on the state of the vehicle as well as the environment and the control actions taken. It is filled by the *SensorEvaluator* classes and read out by the *Observer*.
- **Sensors Evaluators:** These classes read out sensor values by subscribing to their ROS topics and processing the information gathered to create meaningful information from them, i. e. detecting obstacles or lanes. The processed information is added to the *Model*. To gain information from a sensor and put it into the model, this class needs to be inherited from.
- **Observer:** This class acts as an Observer in the context of control design. The data in the model only represents the observable state, which may not be the complete state information needed to control the system. The observer estimates the actual state from the observable model. The *Controller* and the *Visualizer* read from this observer instead from the *Model* directly. If the model is amended, the observer probably has to be altered as well.
- **Controller:** Reads the state information provided by the *Observer* and decides on a control action depending on that state. Writes the control action back into the *Model*. Adaptation of the robots actions is done here. Third parties are able to write their own controllers, in order to implement driving functions.
- **Visualizer:** Reads the state information provided by the *Observer* and visualizes it through a GUI. Additional visualizers may be added.
- **Transposer:** The goal is to simulate a car which has a different behaviour than the TB. The Transposer reads the control actions from the *Model* and maps them to the behaviour of the car model. It then publishes messages via the *TurtleCarNode* to the bot's actuators so that the robot shows that behaviour. Since it simulates the car, it also writes the information about the car's new state - like the current gear - back into the *Model*.

7.1.2 TurtleCarNode Core Loop

The core loop of the *TurtleCarNode* on a high level is shown in Figure 7.2. The node starts with initializing the model, observer, transposer, sensor evaluators, controllers and the visualizer. After that, timers are created for the different tasks TurtleCar-Core fulfils. Timers are used to ensure that different tasks can be scheduled for different time intervals.

The different tasks TurtleCar-Core fulfils are shown in table 7.1

ROS guarantees that timer callbacks and sensor callbacks don't execute simultaneously. Only one callback is processed per loop, preventing any concurrency issues.

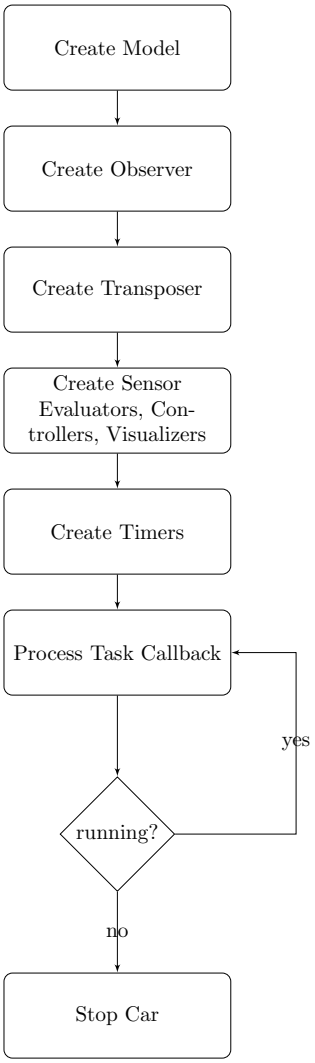


Figure 7.2: Start and core loop of *TurtleCarNode*

Task	**Interval**	**Description**
Plotter	1 s	Updates the plotter for the EKF
Main	0.1 s	Updates observer, controller and transposer
Prediction	0.01 s	Predicts next state
Visualizer	1/30 s	Redraws the visualizer and process inputs
Model Broadcast	0.1 s	Publishes parts of the vehicle state for external input/output devices
Platoon Broadcast	0.2 s	Publishes own position and detected obstacles to the platoon members (see Section 16.7)

Table 7.1: Tasks of the *TurtleCarNode*

7.1.3 Filtering Sensor Values

The information gathered from the sensors via the ROS interface may need to be filtered to be usable by the modules interpreting that sensor data. In the context of the project group, two variants of filtering are defined, which are reflected in different aspects of the architecture:

Technically Necessary Filtering There exist technical reasons for filtering values directly when they are retrieved from a ROS message. One example is the LIDAR: It has a varying resolution which must be upscaled to a fixed resolution by interpolating missing values. This is done directly when retrieving the values. Sensor Evaluators using the standard `lidar.subscribe_lidar()` function implicitly receive the fixed-resolution values. When necessary, a similar standard filtering behaviour may be implemented for other sensors as well.

Task Specific Filtering Some Sensor Evaluators may have requirements for filtering the sensor values that are not necessary for processing the values, but are functional requirements related to their task. These filters are implemented in the context of the Sensor Evaluator and only used to fulfill its task, but do not influence the input to other Sensor Evaluators. Each Sensor Evaluator has to explicitly implement the filters it needs or explicitly use a filter function shared between evaluators.

7.1.4 Unit Testing

pytest [43] is used to perform unit tests. For mocking, *mockito-python* [58] is used. When writing unit tests, the following criteria should be met:

- Test one specific aspect of the code under test
- Mock the complete environment of the function. Everything that is not part of the code under test should not be executed.
- If the tests or the mocking effort is high, consider refactoring the tested code to enable smaller tests.

All tests are located in the `tests` directory.

7.2 TurtleCar-Core Coordinate System

TurtleCar-Core uses two different reference frames to use coordinates: a local frame and a global frame.

7.2.1 Local Coordinate System

ROS sends its coordinates inside a local frame where the X-axis is aligned with the vehicle's heading. To the left of the vehicle is the positive side of the Y-axis and to the top is the Z-axis. In the following, the Z-axis won't be mentioned anymore because the scenarios of this project all assume a flat street without any elevation.

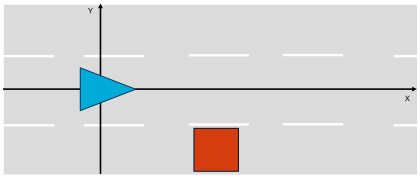


Figure 7.3: Local Coordinate System: Start Point

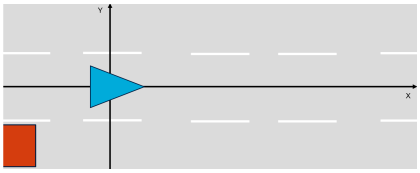


Figure 7.4: Local Coordinate System: Driving

While driving, the origin of the coordinate system follows the vehicle. This kind of coordinate system is useful for calculating the relative distances or angles to other objects.

When the heading of the vehicle changes the coordinate system also rotates.

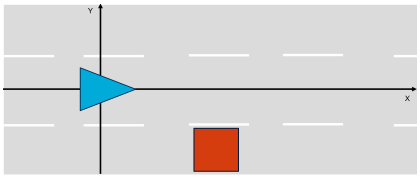


Figure 7.5: Local Coordinate System: Not Rotated

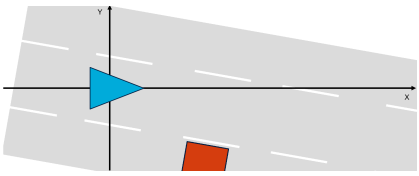


Figure 7.6: Local Coordinate System: Rotated

With a local coordinate system it is not possible to drive along trajectories as the origin keeps moving. This is, however, necessary for control strategies like MPC.

7.2.2 Global Coordinate System

A global coordinate system uses an origin which is fixed in space. TurtleCar-Core uses the start position of the ego vehicle as its first origin. The key difference to the local coordinate system is that even when the vehicle moves the origin stays at its place.

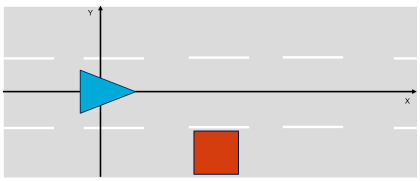


Figure 7.7: Global Coordinate System: Start Point

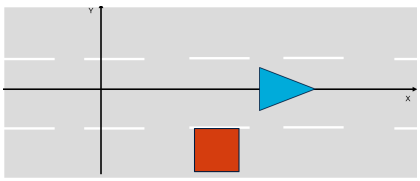


Figure 7.8: Global Coordinate System: Driving

If the position of the vehicle is rotated at startup, the street does not align with the X-axis. This does not pose a functional problem, but can be tricky when trying to debug wrong coordinates.

To ensure the coordinate system is aligned with the lane borders, TurtleCar-Core moves the origin after the first detection of the lane borders. The origin is

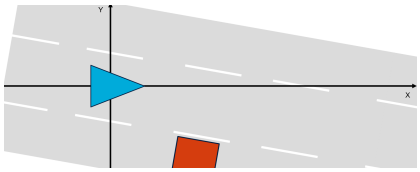


Figure 7.9: Global Coordinate System: Rotated Start

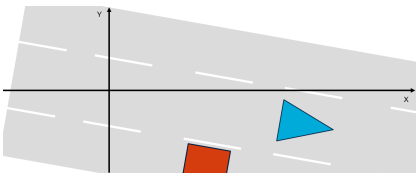


Figure 7.10: Global Coordinate System: Rotated Driving

set on the rightmost lane border and the coordinate system is rotated so that the X-axis is parallel to the street.

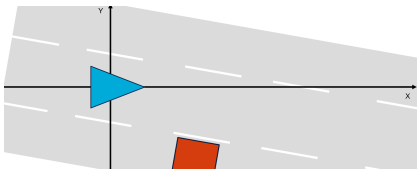


Figure 7.11: Global Coordinate System: Alignment Start

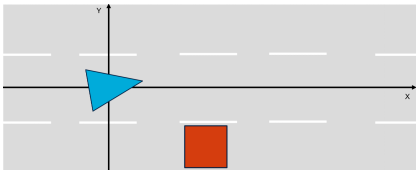


Figure 7.12: Global Coordinate System: Rotate Origin

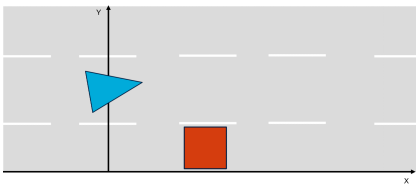


Figure 7.13: Global Coordinate System: Move Origin

The origin of the global coordinate system can also be set from the outside using a ROS topic to send a coordinate and rotation relative to the vehicle. This can be used to unify two or more coordinate systems of different vehicles inside a platoon.

The local lane and obstacle coordinates are globalized every observer update by using the heading and global position of the vehicle state. First the local coordinates are rotated by the heading and then they are moved by the position of the TurtleBot.

7.3 TurtleBot ROS2 Image

It is possible to automatically build a minimal, customized image, which is real-time capable, for the TB. The repository which can be used for this can be found in the project groups GitLab [33].

It is important to note that the image cannot be built using the docker image, due to limitations of *systemd-nspawn*.

Dependencies are customizable by opening

```
image_builder/data/jammy-rt-hubmle/scripts/
```

and adapting the file `phase1-target`. Under the comment „user-specific dependencies“ it is possible to add desired dependencies via apt.

To build the image, change to the top folder and run

```
make jammy-rt-ros2
```

After that, a fully bundled `.img` file is generated, which can be burned to an SD card. Detailed documentation can be found in the repository. Please note this is an updated version of another public project called *Raspberry Pi image with ROS 2 and the real-time kernel* [27].

7.4 TurtleBot ROS2 packages

In the ROS packages provided by the TBs manufacturer, some calculations for the ROS `/odom` topic were missing. Incorrect time comparisons were made here, which led to a falsification of the values of the `/odom` topic. The bug has been fixed and builing the ROS packages manually offers more flexibility in the future, if changes are necessary for the packages.

7.5 TurtleBot Bringup

To simplify starting all services on the TB hardware, a startup script was built which starts all necessary services of the TB at once. Previously, several connections to the TB had to be established in order to start all services. All logs are stored in corresponding process IDs (PIDs) paths and thus make it possible to trace the logged behaviour of the TB. The script also offers the possibility to shut down all started services completely.

7.6 ROS2 WiFi network with TurtleBots

This section aims to provide a overview of Zenoh and integrating it with the TBs and client devices.

Zenoh offers significant advantages in terms of network traffic reduction, particularly in scenarios where network efficiency is paramount. Its ability to optimize local processing, employ content-based routing, and filter/aggregate data makes it a valuable tool for minimizing unnecessary network traffic. One key aspect of Zenoh's traffic reduction capabilities is its operation, where devices can process data locally before transmitting it over the network. By intercepting and optimizing local traffic, Zenoh reduces the volume of data transmitted across the network, thereby alleviating congestion and improving overall network efficiency [5], [18].

One of the primary difficulties for ROS over WiFi networks is the inherent unreliability of wireless communication due to multicasting because of the shared collision domain [96]. WiFi networks are susceptible to interference, signal attenuation, and dynamic channel conditions, leading to unpredictable packet loss and latency variations. In traditional multicast protocols, such as UDP-based multicast, these issues can result in significant data loss and delivery delays, undermining the reliability of communication in ROS systems.

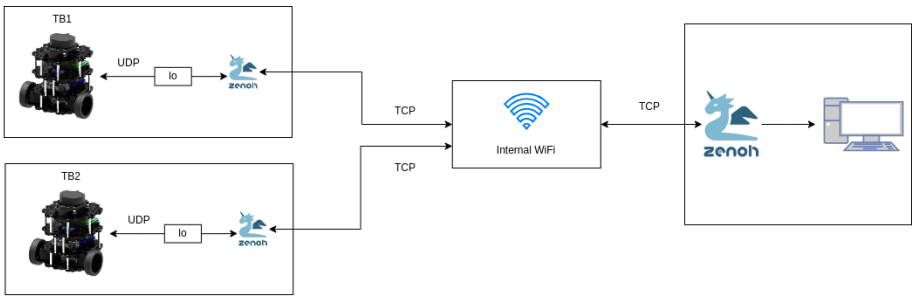


Figure 7.14: An abstract representation of the network structure when using Zenoh.

During the PG, it was repeatedly noted that there was a large dropout of data via ROS. This became even worse when two TBs were operated simultaneously. The clients received less and less data until finally almost nothing arrived and the bots could only be reached via an SSH TCP connection.

ROS multicast utilizes the Publish/Subscribe (Pub/Sub) messaging paradigm, where nodes publish messages to specific topics, and other nodes subscribe to receive messages on those topics. Multicasting in ROS involves broadcasting messages to multiple subscribers interested in particular topics simultaneously. However, traditional multicast protocols face challenges in WiFi environments due to packet loss, latency, and network infrastructure limitations.

Zenoh addresses these challenges by introducing a approach to data distribution that is specifically designed to overcome the limitations of traditional multicast protocols in WiFi environments. By decoupling data producers from consumers and introducing intelligent routing and caching mechanisms, Zenoh optimizes the distribution of data across the network, ensuring reliable and efficient communication in ROS systems over WiFi using TCP for bridging the connection [4], [3].

The Zenoh integration process is relatively straightforward. Firstly, on the TBs, the bringup process and all relevant components are initiated. However, instead of transmitting packets to the network, they are directed to the loopback device, `lo`, effectively preventing network transmission. Now, concerning Zenoh, it is launched on the TB, where it detects ROS traffic on the loopback device, which is limited to local communication. Zenoh intercepts this local traffic and makes it available on a TCP port. To receive these packets, Zenoh must also be installed on the clients, such as laptops, to establish a connection with the Bot using port specified port. Once Zenoh is connected, the traffic is received directly on the laptops. An abstract overview of the described behaviour can be viewed in Figure 7.14. It's worth noting that a binary must be started within the Docker development container to interact with the TBs at all. This additional step is a minor drawback to consider when working with the container.

Chapter 8

Code Quality

In this part of the documentation, elaboration on the decisions regarding Coding Style and Static Code Analysis (SCA) can be found. In the context of the project group, Coding Style and SCA are differentiated. Coding Style includes the ruleset and principles which influence what code is produced. SCA consists of development tools and CI/ CD methods, which allows maintaining parity to the set Code Quality. CI stands for Continuous Integration, and is a typically automatically triggered process that performs tasks such as checking the source code, running tests and ensuring that the source code is compilable [26]. With this process, the goal of having a stable code repository in terms of Code Quality is supported, since automatic Code Quality checks are possible. This is explained in more detail in Section 8.3.

8.1 SCA

There are two main parts of SCA used in the project group: formatting and linting. The coding style is provided by the tools used.

Formatting is the way how the code is formatted: which indentation size is used, how long lines should be and where newlines are located. Formatting ensures that each line of code that is written is in a format that is comprehensible by each member of the project group. The code formats automatically and no further manual intervention is required.

Linting on the other hand makes sure that the code that is written is error-free and adheres to a certain code style: Here, checks against unused variables, long lines and unnecessary complexity are employed. Some linting errors are also fixed by formatting, i. e. long lines. But because fixing most linting errors is a non-trivial task, oftentimes manual intervention is required.

8.2 Development Tools

In order to ensure that the code is in the correct format and to reduce its errors, tools are employed. For formatting, use *black* [8] is used. For linting, use *ruff* [74] is used.

Both tools were chosen for the following reasons:

- Opinionated
  - Being opinionated allows for adhering to community rules forged by years of development time.
  - At the beginning, there is no desire to employ custom, project group specific rules. Everything is changing constantly - there is no need for complex configurations, but for quick usage.
- Modern
  - Modern tools allow for staying cutting-edge.
  - They improve the readability of the codebase.
- Fast
  - Being fast means every machine can run the tools, even if one developer happens to have a slow machine (by modern standards).
  - There is no need to worry about the code base growing so large that the SCA tools will take an unreasonable amount of time to run.
  - Limited numbers of job runners are available in GitLab, as the instance is self-hosted. Therefore, being fast reduces the occupation on those limited resources.

8.3 Continuous Integration

In this section, the ways CI is used in the project group are described. Also, the configuration of the employed pipelines is explained. Pipelines are essentially a set of steps the source code has to pass in order to be valid.

8.3.1 Integration in the workflow with CI

In order to allow constant integration, *black* and *ruff* are used not only locally, but also in the GitLab projects pipelines. This ensures that every commit and merge request is checked.

If *black* detects that the format is not correct or *ruff* finds any linting errors, merging the respecting merge request is disallowed. Also, reviewers will immediately take notice of this and will ask the developer to fix this.

This ensures that the code in the stable branches of the projects remains protected and in a valid state. Additionally, this provides fast feedback for developers whether their code contains errors. This makes locating and fixing errors faster.

8.3.2 Pipeline

In the pipeline, ruff and black are executed. In the following, the mechanics of the pipeline are documented. This part explains the following:

- Elaboration on the pipeline concepts, not the details
- Explanation of the most important caveats, like caching and sometimes allowing pipes to fail
- Starting point for getting to know the pipeline

How the pipeline works The following will explain the structure & concepts of the pipeline in use. In the implementation of the pipeline configuration, the official Python docker image is used, so that some configurations for the executing runner are already present.

Pipeline Building Blocks

- `before_script` Block
  - Ensures that a virtualenv is used
  - Debugs the Python version
  - Executes before each job
- `build-job`
  - Currently only a stub
  - Might be used later, when actual building of the ROS packages is required
- `format-test-job`
  - Runs black and checks for formatting errors
  - Prints encountered errors to 'stdout' for debugging purposes
- `lint-test-job`
  - Runs ruff
  - Looks at the .pyproject.toml file in order to configure ruff
  - Generates a codequality artifact .json, which is used by GitLab to measure code quality
  - Also prints all encountered errors and warnings to `stdout`
  - By using dependencies, this job only runs after `format-test-job`

Important note: The `test` in the jobs name refers to the task of testing if the source code is in a conforming state. This does not mean that the jobs are only 'test' versions.

Caching the installed pip packages The `cache` is used in order to let the runner cache installed packages, so that ruff und black are not reinstalled in every run of the pipeline.

By configuring `PIP_CACHE_DIR`, pip is told to cache its dependencies and installed packages in the directory provided - which are defined as a pipeline cache directory as well. Therefore, the cache directory gets cached in between job runs and reused.

Conclusion: Working with the pipeline Now that the pipeline is configured, it is possible to review Merge Requests based on their generated code quality report. Also, this makes sure that every line of code is formatted in a consistent way. When committing to a custom branch, or merging to 'main', the pipeline is evaluated and run. Project members are required to provide conforming source code, and get hints to why their changes might not be of the desired quality.

Chapter 9

Lane Detection

For a vehicle used in the context of autonomous driving, the ability to perceive and understand its road environment is of high importance. One crucial aspect of this perception is the lane detection, which involves identifying and tracking the lanes on the road. Accurate lane detection is a fundamental building block for many autonomous driving functions, from simple lane-keeping assistance to complex path planning and decision-making algorithms. This section introduces two approaches to lane detection, one based on LIDAR and on based on camera data.

9.1 LIDAR-Based Lane Detection

LIDAR technology plays an important role in the project's implementation of lane detection, as the LIDAR provides essential data about the robot's surrounding. The concept here is to utilize this data to calculate and represent lane boundaries accurately. Visual lanes as indicated by lane markings are therefore not directly detected but are rather projected based on a given road configuration and a rightmost boundary that is detectable by the LIDAR.

9.1.1 Preconditions

The calculation of the lane boundaries using LIDAR assumes that a certain structure for the lanes is always present. One particular assumption is that there always exists a wall that is detectable by the LIDAR sensor on the right side of the road. Furthermore, the first lane always has a distance of w_s to this wall, forming a road shoulder with constant width. Additionally, every lane has the exact same constant width, noted as w_l in the following.

9.1.2 Coordinate Transformation

The process of the LIDAR-based lane detection begins with transforming polar coordinates into the Cartesian coordinate system. This conversion simplifies subsequent processing steps and provides a clear representation of the environment. Based on a respective angle α_i and a distance value d_i of each LIDAR measuring point i, Cartesian coordinates x_i and y_i for such point can be created using the common formulas $x = d \cdot \cos(\alpha)$ and $y = d \cdot \sin(\alpha)$.

9.1.3 Boundary Detection and Lane Projection

Once in Cartesian coordinates, the system calculates the lane boundaries based on the distance of the robot from the right wall.

In particular, this calculation uses the coordinates of the wall that is detected by the LIDAR between 180° and 359°. Not all 179 points are used for the B-spline approximation. With every measurement a 90° cone is aligned to the closest point at right wall. This ensures that at every (realistic) angle the lane can be detected.

A B-spline is then fitted to these data points, ensuring smooth and continuous representation of the lane-defining wall. Using B-splines offers the possibility to control the degree of the lane boundaries, which is useful to extend the lane projections from straight to curved roads. For individual points of the given B-spline, normalized orthogonal vectors are then calculated. This is done by first calculating a tangential vector of a given point on the B-spline, normalizing that vector and then rotating it by 90° into the correct direction. For a given lane $n \geq 0$, these normalized vectors are used to determine the position of the lane's right ($j = n$) and left ($j = n + 1$) boundary, if multiplied with the factor $(w_s + j \cdot w_l)$.

To ensure robust lane perception, some additional mechanisms were implemented. Detected lanes are always analysed by their slope and may be discarded if the boundary has a turn in it, which is irregularly steep. This is due to the fact that a steep turn in a lane is an indicator for a faulty calculation that originated from an obstacle at the right side of the ego vehicle, which was mistaken as the boundary wall. To make sure that many subsequent faulty lane calculations do not lead to the model and the reality „drifting apart“, an additional lane detection switch is added. If multiple faulty lanes are detected in a row, the direction at which the boundary wall is assumed is switched and the respective calculations are adjusted. This way, correct lane boundaries can still be obtained, even when the ego vehicle is passing an obstacle.

The LIDAR-based lane detection is simple and provides good results if the detected Lane is straight. However, steep curves pose a problem, as they can not be reliably detected using the presented spline fitting approach. Also, even with the side-switching, the method is not reliable if the sight line from the LIDAR to the lane borders is obstructed. Therefore, another approach to detecting lanes is required in order to compensate for these deficits.

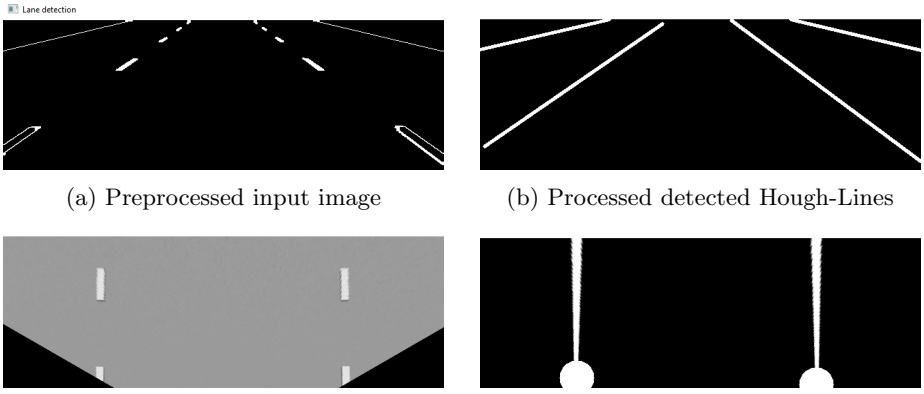
9.2 Camera-Based Lane Detection

Limitations of the LIDAR-based lane detection approach, such as the requirement of a wall being present on the lane boundary, have motivated the implementation of a camera-based alternative approach. Camera-based lane detection allows the vehicle to detect lane markings using visual data from its onboard camera.
The project group has explored two distinct methods for lane detection in camera images: classical computer vision techniques and an AI-driven approach. The latter, overcoming limitations of the former, has been adopted.

9.2.1 Classical Computer Vision Approach

The initial approach for the detection of lanes was based on the Hough Line Transform, which is a technique for detection of straight lines within images [39]. This method followed several stages as seen in Figure 9.2 (outputs are visualized in Figure 9.1):

1. Region of Interest (ROI) Segmentation: Isolation of the road segment from the camera's field of view.
2. Preprocessing: Conversion to grayscale, blurring, edge detection via the Canny algorithm and morphological closing to prepare the image for the Hough Transform.
3. Hough Line Transform: Detection of potential lane line candidates in the preprocessed image.
4. Postprocessing: Filtering and clustering of the lane line candidates.
5. Bird's-eye View Transformation: Transformation of the coordinates for further use.
6. Lane Line Extension: Scaling and extrapolation of the detected lines.
7. Lane Data Calculation: Estimation of the vehicle's current lane and the positions of adjacent lane boundaries.



(a) Preprocessed input image

(b) Processed detected Hough-Lines

(c) Bird's eye view of the input image

(d) Bird's eye view of the Hough image

Figure 9.1: Outputs of the stages of the initial approach.

The segmentation ensures a focus on the road, while the preprocessing steps reduces noise and filtered unnecessary information from the image to ensure better performance of the Hough Line Transform. Since the Hough Line Transformation has a tendency to detect multiple lines where there should only be one, clustering and filtering of outliers allows increases the precision of the detection.

While this approach performs well in predefined simulation scenarios, it isn't sensible in the less refined real-environment setup. Additionally, a large caveat

of it is the inability of detecting curved lines, which is a crucial aspect for the project group, and thus necessities the development of another approach.

9.2.2 AI Enhanced Implementation

In contrast to the first approach, the AI-driven approach solves the problem of lane detection using a pre-trained AI model, specialized in the detection of lane lines on highways. The followed approach, called „Ultra Fast Lane Detection 2“ (UFLDv2) uses a ResNet-18 based model trained on the TUSimple dataset [83], which consists out of 6408 road images on US highways with the resolution 1280×720 pixels. UFLDv2 delivers a high-performance solution to lane detection suitable for the computational constraints of this project. The models' architecture is described by Qin et al. also providing code for it [69], [16]. Initially the implementation was using the first version of UFLD [68], [15]. The switch to UFLDv2 was made as it uses an improved model architecture and generally detects lanes more reliably compared to UFLDv1, as can be seen Figure 9.4.

An important consideration when researching the feasibility of the usage of an AI model for this task was the limited computational power accessible. Since the project requires for the computation to run on a Laptop CPU or on the TB itself, most available AI models are not suitable. A solution for this issue is provided by Kai Chun, offering an implementation of UFLDv2 in the ONNX format [36], [30]. The ONNX version of the model itself can be downloaded from a pretrained model collection [64]. ONNX is an open source library, which amongst other things provides a hardware optimized format for AI models, allowing performant inference on CPUs.
The lane detection process in this approach works as follows (Figure 9.3):

1. Cropping: Cropping of the captured images to the 4:3 aspect ratio which is required for the model.
2. Inference: Passing of cropped images to the AI model, which detects the lane lines and returns their coordinates Figure 9.4.
3. Bird's-eye View Transformation: Transformation of the coordinates for further use Figure 9.7.
4. Lane Line Extension: Scaling and extrapolation of the detected lines.
5. Lane Data Calculation: Estimation of the vehicle's current lane and the positions of adjacent lane boundaries.

The last three steps remained the same as in the initial approach, as they are independent of the actual lane detection method.

9.2.3 Preconditions

For optimal performance, the road texture used for the Gazebo simulation of a highway environment has been updated with a higher resolution one. Additionally, the camera height and angle have been adjusted to match the perspective of

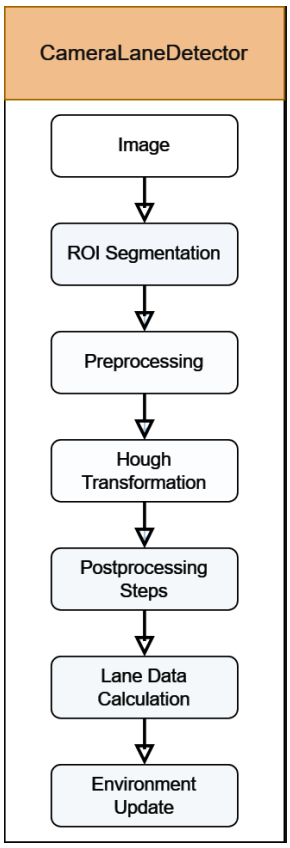


Figure 9.2: Stages of the lane detection in the original approach.

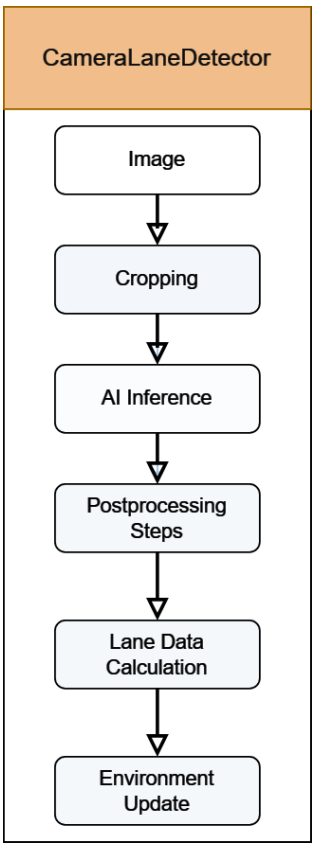


Figure 9.3: Stages of the lane detection in the AI enhanced approach.

Figure 9.4: Comparison of detection using the first and second version of UFLD.

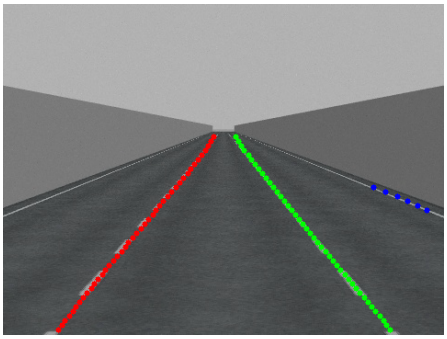


Figure 9.5: The lane lines detected by UFLDv1.

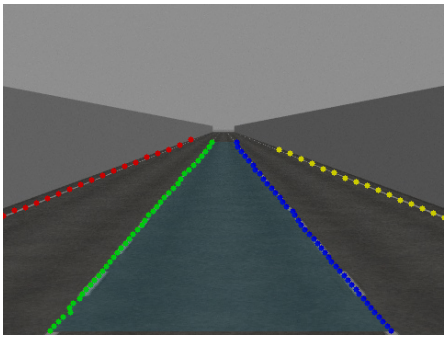


Figure 9.6: The lane lines detected by UFLDv2.

the cameras used for the TuSimple dataset. These aspects influence the model's ability to discern lane lines and present a challenge when using the actual TB, as the real-world camera setup and environment are not as easily adjustable as

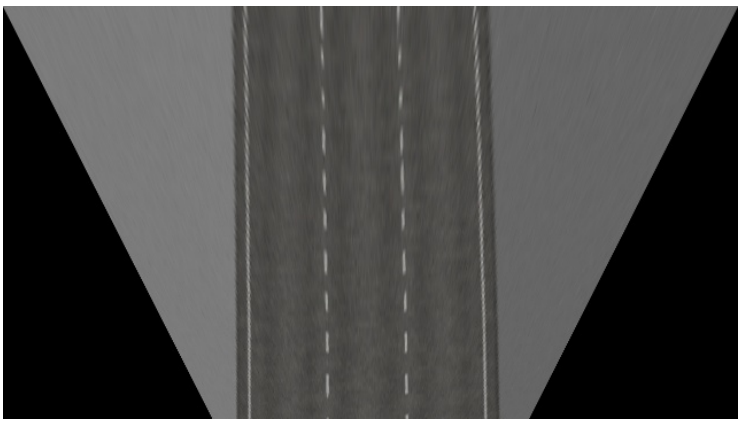


Figure 9.7: The image after the perspective transformation.

the simulated counterparts.

9.2.4 Bird's-eye View Transformation

In order for the lane line coordinates detected in images to be compatible with the internal coordinate system, a Bird's eye view transformation needs to be applied to them. This is done by using predefined source points and destination points for the transformation, which have been manually established using the images captured by the camera. As the real camera and simulated camera differ, the adequate camera lane detection configuration must be loaded. Using the set points, a transformation matrix is computed and applied to the detected lane line coordinates to acquire the Bird's eye view coordinates. The implementation of this functionality is done based on Falaleev [25].

9.2.5 Lane Data Processing

Lane lines below a certain length are filtered, as short lane lines detected by the model tend to be more inaccurate, based on performed experiments. As the camera perceives certain distance in front of the vehicle (varying on the camera's field of view), the lane lines are also extrapolated "backwards". This is done because the lane coordinates at the location of the vehicle are required for further calculations. Finally, the lane line coordinates are rescaled to match the dimensions of the internal coordinate system.

9.2.6 Advantages and Limitations

The camera-based approach is an alternative to the LIDAR-based lane perception and has advantages such as not requiring a border wall next to the road for it to work. Furthermore, it is able to predict coordinates of lane lines which are partially obstructed by objects such as other vehicles on the road, which the LIDAR approach cannot.
However, this approach is not perfect as the nature of the AI model makes using it to acquire reliable lane information difficult. The detection is affected by various factors such as lighting conditions or reflectivity of the road. For instance,

as depicted in figure Figure 9.5 the first version of UFLD did not detect the left lane line, even though it is visible. This inconsistency may originate from the pretrained models training data, which consists of images captured using a camera with a different field of view and resolution. Furthermore, it was found through testing that the lane lines making up the boundaries of the current lane the vehicle is on, are detected most consistently, and the general detection seems to struggle while the vehicle is driving over lane lines (e. g. during a lane change). The environments used in the project deviate from the environments in the models training data, which also likely contributes to the observed discrepancies as the model has limited generalization capabilities.
A consideration to be made is the retraining of the AI model using the same dataset (TUSimple), but adjusted for the available camera. This improvement should allow the model to provide better predictions for the images captured by the camera used in this project and thus increase the quality of the lane detection.

9.3 Current Lane and Relative Position in Lane Calculation

The process of determining the vehicle's current lane and its relative position within that lane is independent of the detection method used, as both the LIDAR and camera-based approaches provide processed lane data in the same format. To estimate the current lane, the system calculates the distances between the vehicle and lane borders using the average distance towards the points along the borders. Consequently, the two lane borders that are the closest to the vehicle allow identification of the index of the lane the vehicle is currently on. Following the estimation of the current lane, the lane width is calculated based on the average distances between the lane border points of the current lane. The width is then used to determine the relative position of the vehicle within the lane by dividing the distance to each lane border by the lane width. The validity of the detected lane is additionally assessed by checking whether it falls within a reasonable threshold relative to the predefined lane width of the environment.

Chapter 10

Object Detection

For a vehicle used in the context of autonomous driving, the ability to detect obstacles within its environment is critical, as obstacles need to be perceived to evade them and drive safely. The TB can perceive obstacles in two ways: through images with a camera and distance measures with a LIDAR. The implementation of both methods is described in the following sections, along with their advantages and disadvantages.

As a simplification, the following assumptions are made for static obstacles:

- Static obstacles are as wide and as long as the lane's width, so the unscaled dimensions are $0.3\,\mathrm{m} \times 0.3\,\mathrm{m}$.
- Static obstacles are always positioned on exactly one lane, conversely a static obstacle has to be positioned parallel to the lane it is on.
- In between two static obstacles there is at least one obstacle width of free space.

10.1 LIDAR-Based Obstacle Detection

In the context of LIDAR-based obstacle detection, there are various existing approaches. Here, the focus will be on LIDAR obstacle detection based on filtering, segmentation, and clustering of point clouds created from two-dimensional LIDAR data. Such an approach is for example described by Likhita et al. [49], Soitinaho et al. [81], and Peng et al. [62]

Peng et al. [62] introduce an obstacle detection and obstacle avoidance algorithm based on 2D LIDAR by filtering and clustering the laser-point cloud data. Their approach is divided into three steps. First, they filter the raw LIDAR data. Then, the result is preprocessed: the laser-point cloud is segmented into several laser-point clouds and, next, merged if the clouds are close to each other. In the end, each cloud gets a shape assigned depending on the points they contain. They differentiate between circles, rectangles, and lines. For further insights, Mochurad et al. present a comprehensive overview of different downward- and horizontal-looking 3D and 2D and LIDAR obstacle detection approaches and their advantages and disadvantages [57].

The LIDAR obstacle detection for the TB is based on the approach by Peng et al. [62].

For the LIDAR-based obstacle detection, the 360 Laser Distance Sensor LDS-02 is used. This 2D LIDAR detects distances 360 degrees around itself. Each new scan contains up to 360 scans and information about the angle increment between each. Since the simulated environment always returns a value for each degree, but the real-world LIDAR does not, there is a need to handle the disparity. Using interpolation the real-world data is augmented to assure it contains 360 values. The LIDAR is placed horizontally on the TB and can therefore only detect obstacles as high as the TB. Any obstacle smaller than the TB can only be detected using the camera.

An obstacle can only be detected if it is not hidden behind another obstacle. Furthermore, other vehicles must also have a LIDAR at the same height as the detecting vehicle to be identifiable. The identified LIDAR's shape must also have a set radius of 0.03 meters (unscaled).

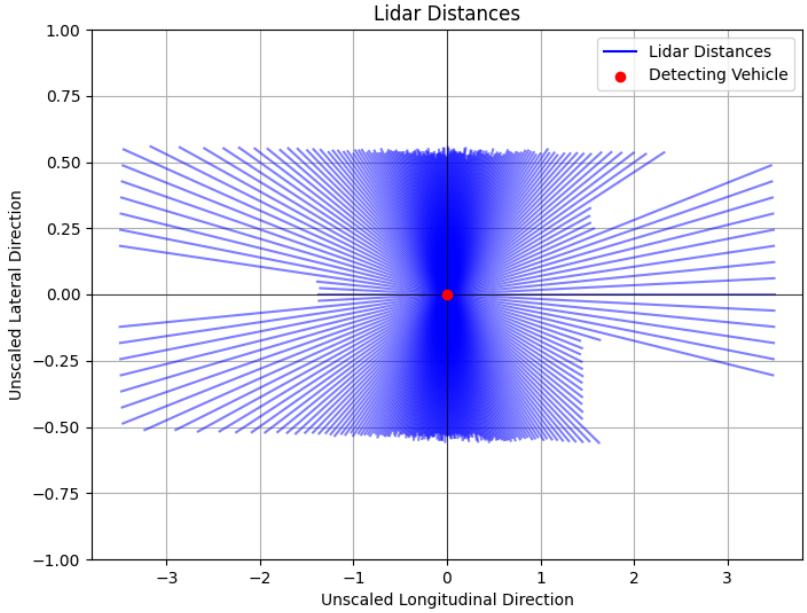


Figure 10.1: LIDAR Data

LIDAR operates by emitting laser pulses detecting their reflections from objects. If there are no objects obstructing its path, each pulse is of the same length. However, if there are objects, the lengths differ. This can be seen in the example depicted in Figure 10.1.

The vehicle is placed in the middle of the road and the LIDAR mounted on top of it detects five different barriers. Considering the vehicle is heading along the x-axis, on the left and right of the detecting vehicle are the road walls. Directly behind the detecting vehicle, there is a vehicle, and another one is front left of it. A static obstacle is located at the front right.

The LIDAR-based obstacle detection needs four steps to process the raw LIDAR data into an interpretation like exemplarily done above. These steps

are depicted in Figure 10.2, Figure 10.3, Figure 10.4 and Figure 10.5.

Initially, the LIDAR data has to be preprocessed since it only provides distance measurements in different directions. When receiving LIDAR data, it may not always contain a distance for every degree, resulting in a list of varying lengths but at most 360 values. To address this variability, this list is linearly interpolated first as explained before. This way, the obstacle detection can always expect 360 values whether used in simulation or in the real world.

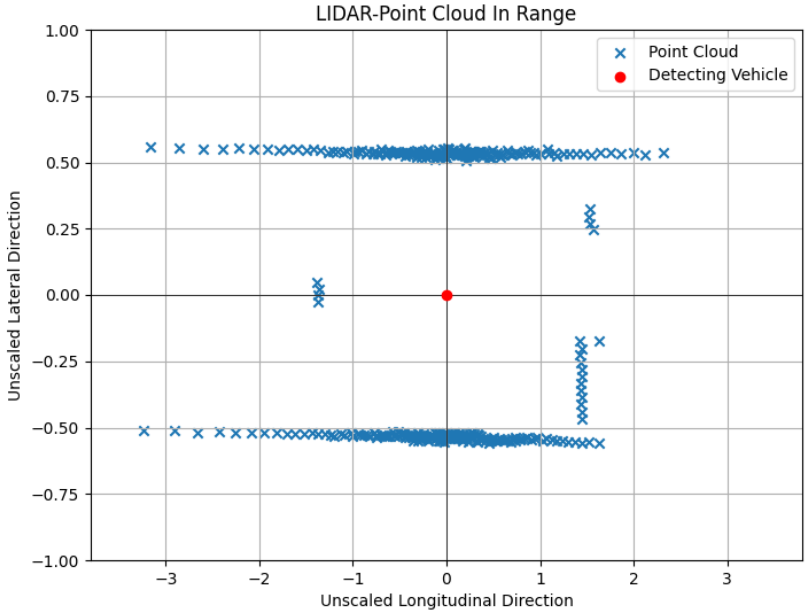


Figure 10.2: LIDAR-point cloud in range

The 360 distance measurements are then converted into points relative to the vehicle's position as seen in Figure 10.2. Since the LIDAR can also detect the street wall (if there is one), some of the points might represent the wall and not an obstacle on a lane. Because of this, points that are on the edge or outside of the street are removed. This is depicted in Figure 10.3.

The remaining points are divided into clusters defined by the distance between the points. If two points are at most a set threshold distance apart, they are put into the same cluster. The threshold distance is chosen as the width of a vehicle. Following, points the vehicle can not drive in between belong into one cluster and are seen as one obstacle. Using the example from Figure 10.3, the result would be three clusters as seen in Figure 10.4

Each resulting cluster forms a potential obstacle. In the next step, a circle-fitting algorithm is used to determine what kind of obstacle the cluster represents. The cluster should either fit the dimensions of the LIDAR on top of the other vehicle or the dimensions of a static obstacle. If it does not match either, the obstacle is detected as an unknown obstacle type. Additionally, if the detected obstacle is static, it is further evaluated whether the obstacle is

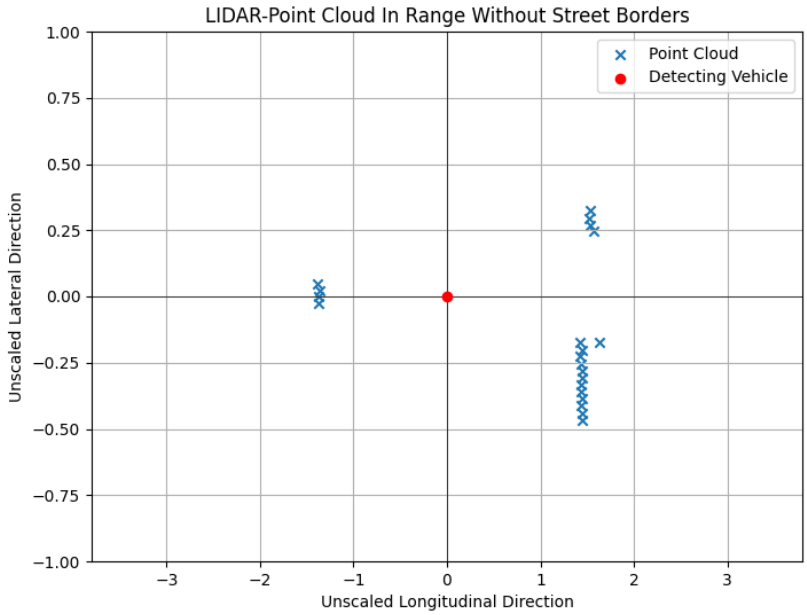


Figure 10.3: LIDAR-point cloud in range without street borders

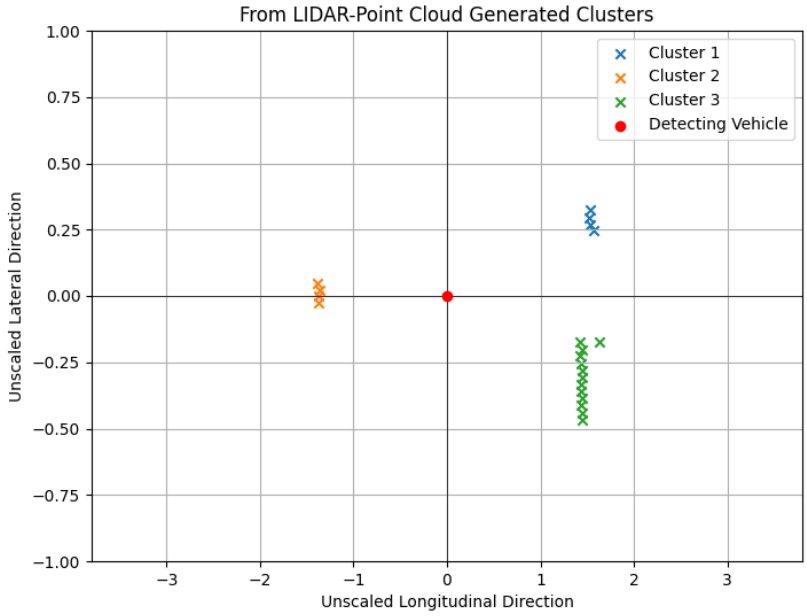


Figure 10.4: From LIDAR-point cloud generated clusters

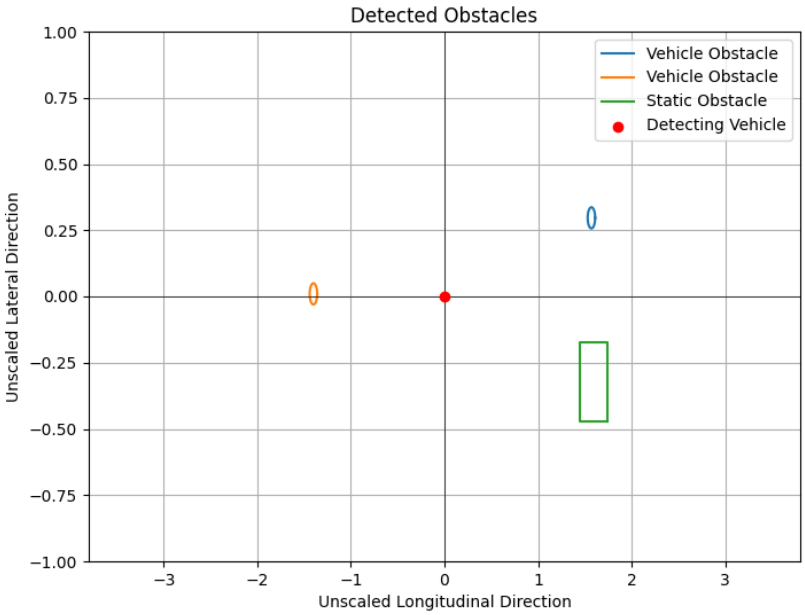


Figure 10.5: Detected obstacles

seen from the front (as a line) or from the side (as two lines). Detected and identified obstacles are then added to the currently detected obstacle list. This step's result is represented in Figure 10.5, where three obstacles have been detected with LIDAR-based obstacle detection: one static obstacle and two other vehicles. Since the figure's axes are not equal, the obstacles' form is seemingly distorted.

Finally, the environment model is updated with the newly detected obstacles.

10.2 Camera Based Object Detection

Various ways of camera-data-based object detection have been researched and evaluated as a part of the project. Considerations included training an object detection AI model (such as YOLOv7) or using traditional computer vision algorithms. The AI based approach, often involving convolutional neural networks (CNNs), is known for achieving high accuracy in object detection. However, its complexity and (usually) resource-intensive nature makes it less ideal for the scope of this project. Traditional computer vision algorithms offer a less computationally demanding alternative, but they might require additional conditions to be met.

This project uses a classical approach, enabled by the fact that full control over the environment and the vehicle is available: Fixed synthetic markers with a known layout. Strategic placement of such markers on to-be-detected object ensures optimal visibility and ease of detection. The simplicity of this approach translates into faster processing on the TB, making it an efficient solution.

Moreover, the required preparation of the markers on the detectable objects leads to the natural elimination of possible false-positive object detections.

10.2.1 Marker Systems

During research of object detection using fixed markers, two prominent state-of-the-art marker systems came into focus: AprilTag [2] and ArUco (ArUco) [61]. Both methods employ square markers, based on a visual bit representation of unique ID patterns. They facilitate rapid and accurate identification, making them suitable choices for applications with limited resources.

AprilTag markers are slightly more complicated to generate than ArUco markers, however, pre-existing repositories that contain different AprilTag-families eliminate the need for individual marker generation. Additionally, the detection of AprilTag markers is implemented in a ROS package, which made it seem like a preferable choice considering that the TurtleCar application already uses ROS. However, the AprilTag package is not fully migrated to the second version of ROS, which complicates its actual utilization and would require additional debugging. On the other hand, the ArUco marker system is included in the OpenCV python package, which is already used by the TurtleCar application. Integrating the generation and detection of ArUco markers was therefore straightforward for the existing architecture and was the preferred choice.

10.2.2 ArUco Marker

The ArUco library allows the detection of ArUco markers along with their distances. The requirement of making the marker detection as reliable as possible, rendered the ArUco marker family `DICT_4X4_50` the most promising. 4×4 indicates the bit size, whereas 50 is the number of available markers in that family. Choosing the minimum in both regards ensures a maximum Hamming Distance between the marker IDs and better overall detectability.

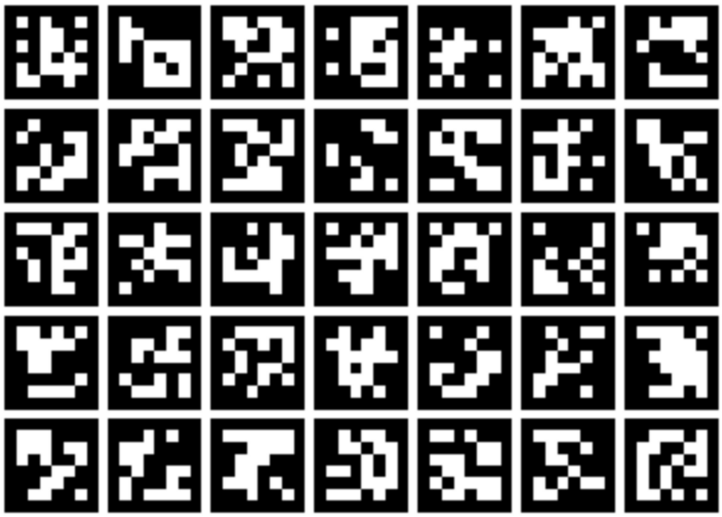
10.2.3 Environment Preparation

For the successful usage of fixed markers, preparation of the controlled environment is necessary in both simulation and reality. For that, all marker IDs available in the chosen marker family are mapped to an object type:

- IDs $0 - 9$: Other vehicles
- IDs $10 - 39$: Cuboid obstacles
- IDs $40 - 49$: Road signs
  - 40: No overtaking
  - 41: Overtaking allowed
  - 42: $80\,\mathrm{km/h}$ speed limit
  - 43: No speed limit

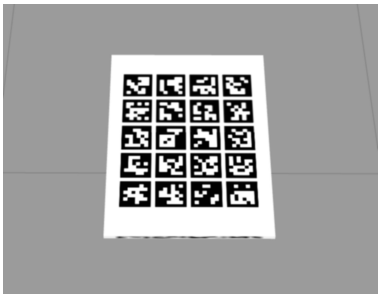
The placement of markers follows a respective strategy for the different object types. Vehicles are equipped with a single marker positioned on their backs,

serving as a distinctive „license plate“. Cuboidal are marked with four distinct markers, placed on each corner of their detectable faces. This enables the calculation of the obstacle's size and rotation along with the distance. For the current scenarios of the project group, these calculations are not strictly necessary since an obstacle in a lane results in a totally blocking that lane independently of its size and orientation. However, this may be useful for future driving functions. Road signs use single markers and are placed on an elevation along either the left or right side of the road.



(a) ArUco board used for camera calibration.

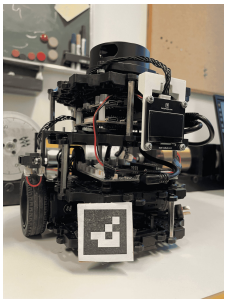
(b) Example image used for the camera calibration in reality.



(c) Example image used for the camera calibration in simulation.

Figure 10.6: Camera calibration for ArUco detection.

To achieve optimal precision in both the simulated and the real environment, an initial one-time camera calibration process needs to be conducted. This process obtains camera parameters, which can then be utilized for camera based detection tasks. For the calibration, a special board with ArUco markers that can be seen in Figure 10.6a is used. With a set of roughly 50 images that capture the calibration board from different angles, the ArUco library is used to acquire parameters of the camera, which are then saved to a config file for further usage. Examples of such images can be seen in Figure 10.6b and Figure 10.6c.

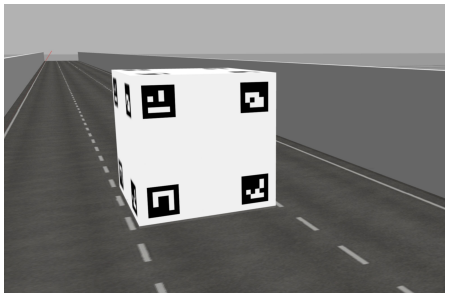


(a) TB marker in reality.

(b) Obstacle marker in reality.

Figure 10.7: Marker layout in real environment.

(a) TB marker in simulation.



(b) Obstacle marker in simulation.

Figure 10.8: Marker layout in simulation.

Additionally, to allow the object detection to accurately calculate distances to the markers, cohesive marker sizes withing reality and the simulation are required. To determine the optimal marker sizes, markers were experimentally printed and attached to the respective objects in reality. TBs in the real environment impose some restrictions, as their license plate should not restrict any other sensors or the overall mobility of the TB. After the optimal marker sizes were established, fitting 3D models could be created. The resulting layout for the TB markers can be seen in Figure 10.7a and Figure 10.8a and those for the obstacles can be seen in Figure 10.7b and Figure 10.8b.

10.2.4 Marker Detection

The actual marker detection process is mostly independent of whether the environment is simulated or real, apart from the correct camera calibration that has to be loaded. The detection simply uses the available camera images and

leverages the ArUco library to detect all markers in the image. After that, the same library is used to calculate the coordinates of the individual markers. For each one detected, a fitting Obstacle object containing the obstacle type and information about its state is created, and stored in the TurtleCar-Core Model. The data stored in the TurtleCar-Core Model is updated in each detection cycle and handled further by adequate observers and renderers.

10.2.5 Road Sign Detection

In addition to detecting obstacles, the TurtleBot is able to recognize road signs, which are marked with the appropriate markers as outlined in 10.2.3. Currently, the system is capable of identifying four types of road signs. For the real-world implementation, these signs were crafted from cardboard, as can be seen in Figure 10.9. Meanwhile, in the simulated environment, a special world model including of these road signs has been created.

It is crucial for the TurtleBot to maintain awareness of road signs locations to ensure it has knowledge of current road rules. Hence, road signs along with their detected type and coordinates are saved within a obstacle history for a defined amount of time. This way the positions of road signs which move outside of the TurtleBots detection area can still be tracked. The calculation of the coordinates of road signs uses the data obtained from the ArUco detector to determine the position of each road sign within the global coordinate system.

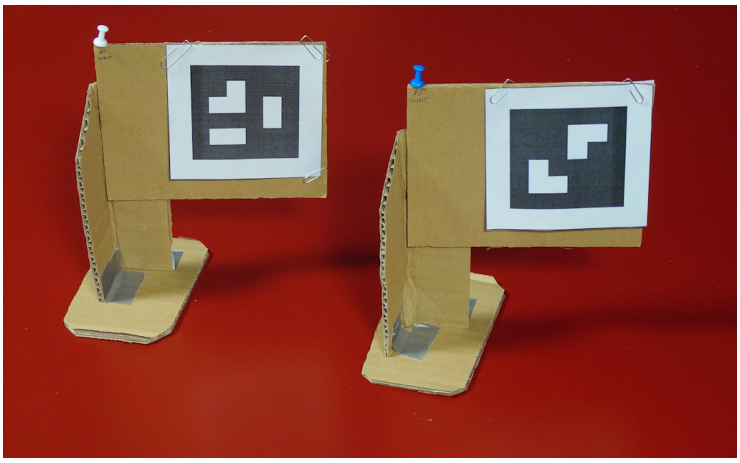


Figure 10.9: Cardboard road signs created for the real environment.

10.2.6 Obstacle Tracking

With the capability to detect obstacles using the camera and LIDAR, a way to keep track of obstacles is needed. This process is intended to not only calculate the speed of each detected obstacle but also estimate its current position, acknowledging sensor errors and minimizing their effects.

Obstacle Identification

A difference between the two detection mechanisms is that the camera assigns IDs to the obstacles it detects based on the marker that was captured. The LIDAR cannot do that as it only detects obstacles based on shape. So while the obstacle identification is already implemented through the ArUco markers, it has to be done independently for the obstacles detected by LIDAR since they initially have no ID.

When the obstacle observer finds an obstacle without an ID it is responsible for the ID assignment. The observer checks already detected and identified obstacles and compares their type and position with the unidentified obstacle. If the positions of the two obstacles are similar enough the observer assumes that both obstacles are the same. When dealing with dynamic obstacles, if an unidentified obstacle occupies the predicted position of a known moving obstacle, the observer also infers that both obstacles are identical. This way the unidentified obstacles are assigned the ID of the already identified obstacles. However, if a similar obstacle can not be found, the LIDAR detected obstacle is assigned a new ID, distinct from all IDs the camera might assign.

This way, both the LIDAR and camera can work simultaneously and in cases where both detect the same obstacle the detections from the camera are kept while ones from the LIDAR are discarded.

10.2.7 Obstacle History

To monitor the states of identified obstacles, a history containing their past positions is maintained. This history allows for determining an obstacle's velocity and comparing its observed behavior against expectations.

The obstacle history is a data structure, that keeps track of previous positions for each detected obstacle. Each entry in the history belongs to exactly one obstacle. Information about the last six positions, the obstacle's ID, and the type of obstacle are saved here. Using this history, the estimated speed of the obstacle is calculated using a sliding window approach.

Due to sensor inaccuracies obstacles might not be detected in every detection cycle of the sensor. However, since obstacles generally don't unexpectedly disappear, keeping track of existing obstacles and filling in missing ones allows for counteracting of sensor inaccuracies. To implement this, the observer keeps track of when the obstacle was detected last and predicts its position in case it is missing unexpectedly. This way it can be re-detected at a later time point and still mapped to the correct previously detected obstacle. Another benefit is that obstacles are still tracked even when they move out of the LIDAR's or the camera's range.

Another approach would be the usage of an EKF to predict the movement of an obstacle. Every obstacle detected would create its own EKF with which the state of the obstacle could be estimated. If the same obstacle would be detected again the EKF could be updated with the new data. If not, then the EKF would return the predicted state. An EKF has the advantage that it would be easy to fuse the information of the camera and lidar obstacle detectors. Due to priorization of other tasks it was decided not to implement an EKF for this use case.

10.2.8 Determining Relative Velocities

By processing detected obstacle data, the relative velocity towards each obstacle is calculated. This calculation allows the identification of moving obstacles and potential hazards for collision avoidance, it works as follows:

1. **Velocity Components:** The x and y components of both the ego vehicle's and the obstacle's velocity are calculated. The velocities are decomposed into their components using the following equations:

$$\begin{aligned} v_{\text{ego},x} &= v_{\text{ego}} \cdot \cos(\theta_{\text{ego}}) \\ v_{\text{ego},y} &= v_{\text{ego}} \cdot \sin(\theta_{\text{ego}}) \\ v_{\text{obstacle},x} &= v_{\text{obstacle}} \cdot \cos(\theta_{\text{obstacle}}) \\ v_{\text{obstacle},y} &= v_{\text{obstacle}} \cdot \sin(\theta_{\text{obstacle}}) \end{aligned}$$

   where v_{ego} and v_{obstacle} are the magnitudes of the ego vehicle's and obstacle's velocity, and θ_{ego}, θ_{obstacle} are their headings.

2. **Relative Velocity:** Relative velocity components between the ego vehicle and the obstacle are calculated.

$$\begin{aligned} v_{\text{rel},x} &= v_{\text{ego},x} - v_{\text{obstacle},x} \\ v_{\text{rel},y} &= v_{\text{ego},y} - v_{\text{obstacle},y} \end{aligned}$$

3. **Vector to Obstacle:** A vector pointing from the ego vehicle to the obstacle is computed.

$$\vec{r}_{\text{to obstacle}} = (x_{\text{obstacle}} - x_{\text{ego}}\, y_{\text{obstacle}} - y_{\text{ego}})$$

4. **Dot Product and Distance:** The dot product is a scalar that measures the magnitude of one vector in the direction of another. It's needed here to calculate how much of the ego vehicle's relative velocity is directed towards the obstacle.

$$\begin{aligned} v_{\text{rel},magnitude} =& (x_{\text{obstacle}} \cdot v_{\text{rel},x}) \\ &+ (y_{\text{obstacle}} \cdot v_{\text{rel},y}) \end{aligned}$$

   $\Delta d = \sqrt{(x_{\text{obstacle}} - x_{\text{ego}})^2 + (y_{\text{obstacle}} - y_{\text{ego}})^2}$

5. **Relative Velocity:** The relative velocity along the line of sight is calculated by dividing the dot product by the distance to the obstacle.

$$v_{\text{rel}} = \frac{v_{\text{rel},magnitude}}{\Delta d}$$

Detected moving obstacles are considered within the alert range d_{alert}, which is currently set to 200 m, if their distance to the ego vehicle is less than the predefined range, and they are moving slower than the ego vehicle. These calculations provide data needed for utilization of various driving functions such as the Lane Change Assistant (LCA).

Chapter 11

Path Planning

In this section, the planning of paths for a vehicle is described. First, Path Planning and Trajectory Planning are defined, and context to these topics within the project group is given. Next, the implementation of Path Planning is explained. Also, the end of the section contains guidance for building and integrating a custom path planning module.

11.1 Definitions and Context

First, Path Planning is different to Trajectory Planning, and an introduction to what both of these terms mean in the context of the project group is given. Further references and mathematical function definitions are given by Pham [63].

Path Planning A path P is a continuous function which connects a start q_{start} and a goal q_{goal} in a coordinate system. Therefore, the domain of P is $[0, 1]$ and its co-domain is $\mathcal{C}$, i. e. , the coordinate space that is used. P is devoid of any time information, and only resembles the geometric component. When enriching it with time information, it becomes a trajectory [63]. For us, planning a path means to plan out a geometric ordered list of points that the robot should follow, disregarding any time information.

Trajectory Planning A trajectory Π is a path P endowed with a time parameterization s. s is a strictly increasing function, which gives the position on the path for each time instant t. Thus, the same path P can give rise to many different trajectories Π [63]. For us, planning a trajectory means to take into account time information to the planned path.

At the current state of the project group, trajectories are not planned, only paths. Planning trajectories would involve many more considerations, which have not been prioritized as of now.

11.2 Implementation

The architectural overview of the path planning implementation can be seen in Figure 11.1. It closely resembles Figure 7.1, but elaborates more on the path planning part.

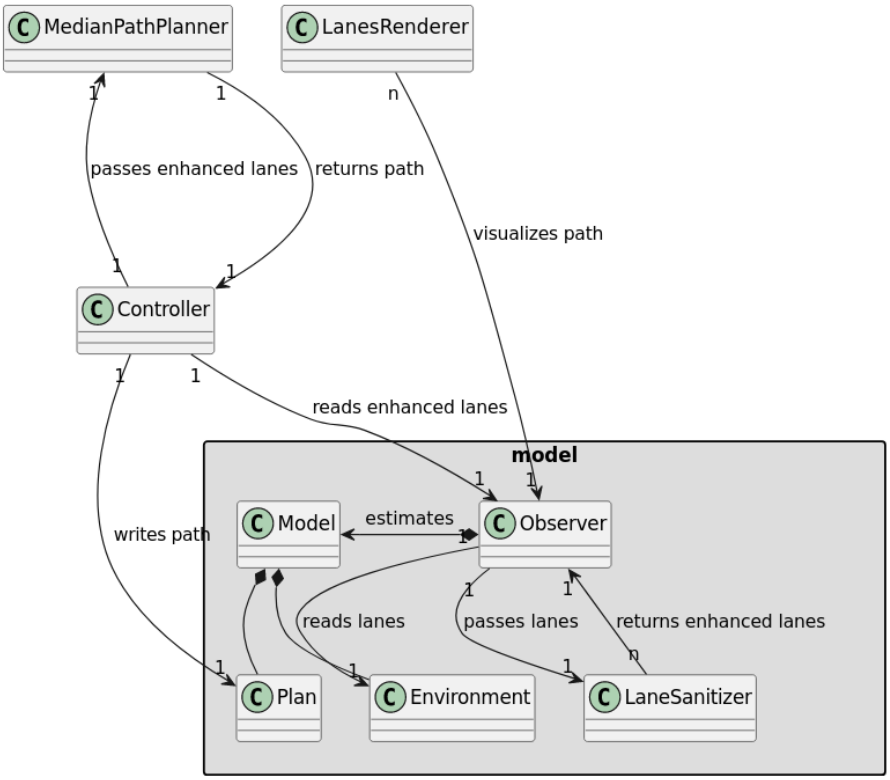


Figure 11.1: Architectural overview with the path planner.

11.2.1 Planning the Path

In order to plan a path, the middle of the border points from the lane data created by the lane sensors is calculated. This creates a path along the center of a lane. The path is visualized via the `LanesRenderer`. See Chapter 9 for more information about how the lane data is created and processed.

11.2.2 Example images

In this section, example images for the path planning module are demonstrated. The snapshots were taken directly from the debugging tool, where orange points visualize paths and yellow points indicate lanes.

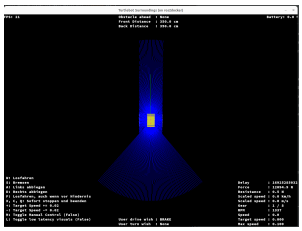


Figure 11.2: An ordinary path

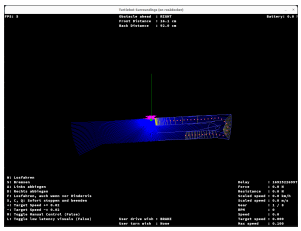


Figure 11.3: A more complex path

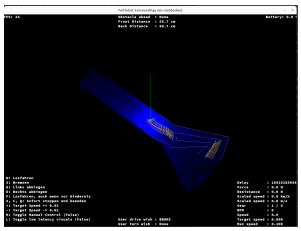


Figure 11.4: Lower border interpolation resolution

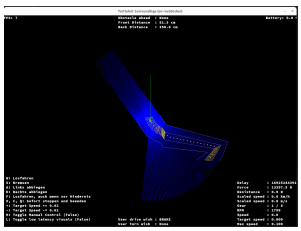


Figure 11.5: Higher interpolation resolution for the border

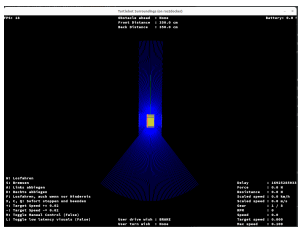


Figure 11.6: Higher sample rate and offset to the left of the border

Chapter 12

Testing driving functions

12.1 Testing Concept

In this secion the approaches used for testing the implemented autonomous driving functions are described. There are many different approaches and methodologies used in state-of-the-art development of autonomous vehicles. First an overview about these is given and put in relation to the project group's approach to validate the driving functions.

12.1.1 Preliminaries

Song et al. collect a number of different available approaches and outline standards and pratices used in industry. Conventional aproaches for testing include unit testing, component testing and code review [82]. These are common methods especially in software development and thus the project group integrated these basic approaches in the workflow (see Section 18.3) and the definition of done (see Section 18.4) for different tickets. However, Song et al.also describe that the approaches are not adequate to capture the complexity of autonomous systems, but should still serve as a starting point for validating parts of autonomy software. The limitations are for example acknowledged by Wang et al. [95]. Methodologies seldomly focus on the whole complexity of autonomous systems, but only on specific functions and provide only limited testing of autonomous vehicles. To overcome the limitations of conventional approaches, more autonomy-focused techniques have been developed. Such techniques include model-based testing, combinatorial testing, search-based testing and scenario-based testing. The first is used to strictly model the system from its operating domain to its behavior, which in turn can be used for test-case generation. The latter two, in contrast, describe approaches, that identify citical scenarios using learning methods. By comparing differing test cases and their results, these can be optimized for challenging test-cases using different criticality metrics and different space exploration methods [82]. Thus such methods are considered to be very promising. Teige et al. from BTC-ES, one of the project group's partners, also describes that test case generation is their most prominent use case. They also recognize that generating test cases based on requirements are of increasing interest [88].

12.1.2 Approach used in the Group

The project group defines its test-cases based on the requirements elicitated for each driving function. For each requirement a test-case can be derived. These requirements however are based on scenarios initially developed by the project group for each driving function Chapter 16. Since these scenarios don't directly provide any requirements and just one general run for that driving function, research had to be done. First, since already established driving funtions were implmented such as the LKA oder ACC, requirements could be derived from literature. Requirements in terms of safety distances could be taken from the StvO. But for other limitations in terms of steering angle for example research was done in different fields like traffic psychology. In conclusion the testing approach can be thought of using a scenario as a baseline and deriving requirement based testing from that.

12.2 Testing with Real TurtleBots

Testing TBs in real-world conditions requires a prepared environment and clearly defined criteria for what constitutes success and failure in these conditions. The project group has conceptually developed two primary methods for conducting these real-life tests, detailed in this chapter.

12.2.1 Approach 1: Fully Manual Testing

For manual testing, the environment is set up to resemble a straight highway. Depending on the driving functions under test, additional modifications to the environment may be necessary. Possible modifications include adequate marking of the to-be-followed path or setting up of obstacles along the highway. For instance, to test the LCA, a path should be marked on the highway that starts in the ego vehicle's initial lane and transitions to an adjacent lane. The ego vehicle must be launched manually, and the LCA activated at the correct moment. The key observation is whether the ego vehicle adheres to the marked path and stays within pre-established boundaries, concluding the maneuver in the intended lane. Success is determined if the ego vehicle meets these conditions. Testing the LKA is simpler and only requires simple markings making clear which lane the ego vehicle should follow. The ego vehicle begins this test slightly angled, after starting to drive and activating of the LKA it should correct its heading to maintain its lane.

Each real-life test scenario requires similarly specific criteria and conditions, which must be manually verified for every test. While straightforward and easy to set up, this method is prone to human error and variability, making it less reliable. Nonetheless, it can offer valuable insights into the vehicles' driving capabilities.

The current real life testing approach employed by the project group follows this approach, albeit without any markings or additional modifications to the highway for simplicity.

12.2.2 Approach 2: Bird's Eye View Camera

This conceptual approach, contrasting with the manual method, allows for a more automated testing process for the vehicles but demands more complex environmental setup. It involves a static camera mounted overhead, capturing the entire highway in its view. This setup allows for an automated testing process similar to that used in the testbed (see Chapter 13), albeit with extra steps required to detect the relevant environment features and objects. The environment captured by the camera must be translated into a digital format, with real-world coordinates matched to those in the simulation. Markers for object detection can be placed on the road borders, obstacles, and the TBs themselves (see Subsection 10.2.1) to facilitate tracking and coordinate matching. If the camera and environment are able to stay fixed in a static position, an alternative would be to use hard-coded coordinates designating the positions of the highway borders and lane lines. This facilitates only needing to track the vehicle with a marker.

This „Bird's Eye View Camera“ setup would enable more autonomous conducting of tests by loading scenarios from predefined files that the real TB would then execute. The overhead camera would monitor the vehicle's performance, comparing its actual behavior against the expected one defined in the according test specification files. For instance, the LKA test could be conducted in a manner similar to the manual approach but observed and evaluated through automated scripts that verify the vehicle's adherence to its lane, providing a more objective and efficient testing solution.

As this is a conceptual approach that has not been implemented by the project group, no infrastructure for usage of automated real life tests exists. However, this concept offers an opportunity for future project groups looking to enhance the automation and precision of real-world TB testing.

12.3 Testing in the simulation

To be able to test developed driving functions more efficiently, the project group developed a dedicated software, called „TurtleCar Test“. Please refer to the next chapter for in-depth information on this topic.

Chapter 13

TurtleCar-Test

TurtleCar-Test (also called Test-Platform) was developed by the project group to test the implemented driving functions on a TB. In collaboration with Gazebo, it allows interactive, headless, scenario-based testing. In this section, related work is presented and then the architecture of TurtleCar-Test based on requirement analysis explained. Furthermore, the creation of tests is explained. Tutorials and examples on its usage can be found in the repository's README [32].

13.1 Traffic Sequence Charts

Traffic Sequence Chart (TSC) proposed by Damm et al. provide a specification language for defining test scenarios for autonomous vehicles [19]. It allows capturing of their behavior in all possible traffic situations. A TSC specification consists of a world model that defines classes of objects (e. g. cars) together with their attributes (e. g. position and velocity) and the dynamics of moving objects, represented by a set of snapshot charts. A symbol dictionary links graphic symbols to their respective objects in the world model. These charts can then be translated to formula in first-order multi-sorted real-time logic.

Snapshot charts describe the evolution over time (e. g. via a snapshot sequence) and are used to visually depict potential traffic situations. They may contain

- present objects,
- relative placements of objects,
- defined absolute distances between objects,
- timing constraints, and
- so-called somewhere- and nowhere-boxes to specify the presence of an object somewhere inside an area or the absence of an object inside a certain area.

Also, they can be composed of premises and consequences. An example snapshot sequence is given in Figure 13.1.

The first two snapshots in the dashed hexagon on the left define the premise, the consequence is specified via the snapshots right of the hexagon. The premise

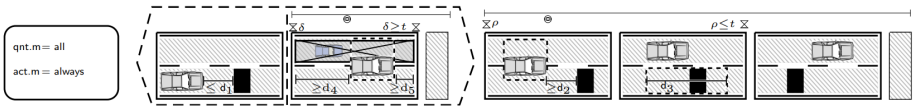


Figure 13.1: Change lane to avoid collision, if the next lane is free [19]

denotes that there is an obstacle in front of a car. Both are on the same lane and less than $d1$ meters apart. From $d4$ to $d5$ there is no car to the lane left of the car (second snapshot). If the premise is fulfilled, the car has to change lanes to avoid collision (last three snapshots).

The TSCs exhibit possibilities to (relatively) place objects, define their attributes (e. g. velocity), define areas and denote premises and expected consequences. These possibilities were also made available in TurtleCar-Test. However, the TSC specification language was not adopted for TurtleCar-Test specifically, since the effort is considered too high while offering no significant advantage in terms of comprehensibility. Even so, in the future, there could be an extension to TurtleCar-Test that allows for the generation of test cases for the Test-Platform from logical formulas generated by TSCs.

13.2 Architecture

Figure 13.2 depicts the simplified architecture of how TurtleCar-Test communicates with the outside world. TurtleCar-Test introduces two main features for writing and executing tests: the Trigger-System (see Subsection 13.2.1) and a Simulated Driver (see Subsection 13.4.4). The Trigger-System receives information from the Gazebo simulation and executes specified actions when specified conditions are met. The Simulated Driver acts as a human driver would and has control over steering and throttle. The Trigger-System and the Simulated Driver in collaboration ensure that the dynamic behavior can be defined in a way that the test requires. These features are further explained in the following sections.

The Gazebo Simulator provides a simulated world, in which the vehicle can be spawned and its condition can be monitored. TurtleCar-Test starts this software at the start of each test and stops it as well, ensuring that no remnants from previous tests interfere with followup tests. With the Test-Platform's Test-Specification-Files, it is possible to define reproducible environments for accurate testing.

The TurtleCar-Core software is also started by the Test-Platform, making it as concise as other testing frameworks like pytest.

13.2.1 Trigger-System

The Trigger-System evaluates data coming from the Gazebo simulation system. It has an internal state containing all Triggers and thereby allows complex testing conditions. The system's structure is visualized in Figure 13.3

The Trigger-System evaluates simulation information from Gazebo based on user-specified Triggers. When writing the testing specification, the programmer

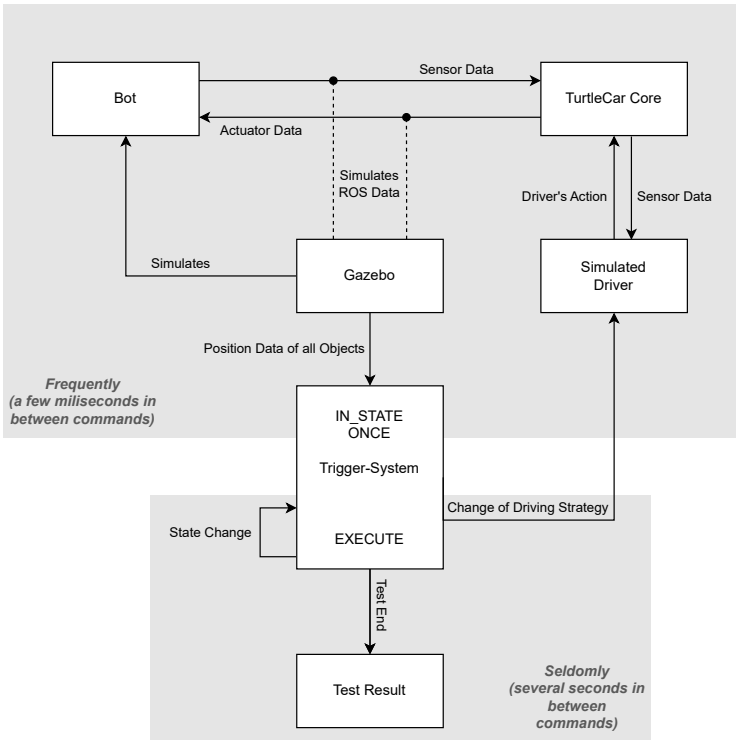
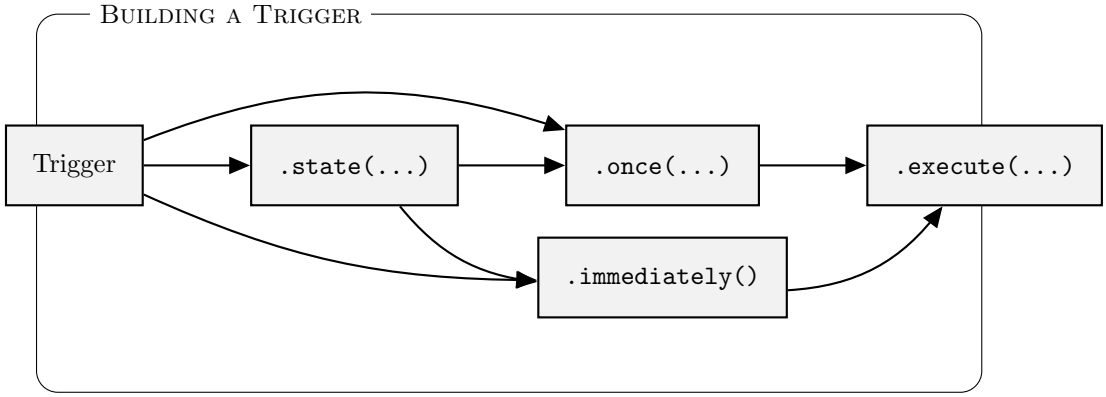


Figure 13.2: Architecture of TurtleCar-Test

can formulate testing conditions and their resulting actions in the form of those Triggers. A Trigger is made up of three components:

1. (optional) State
2. (optional) Conditions
3. Actions

Defining such a trigger is possible in several ways:



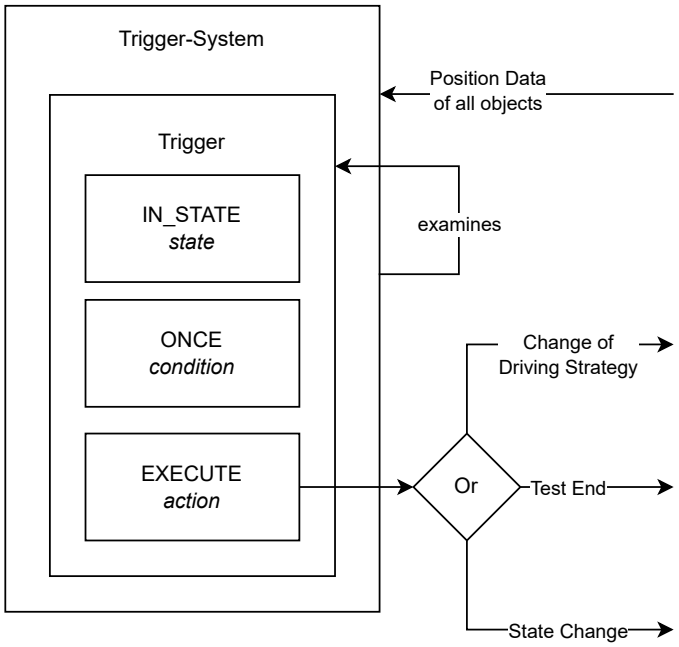


Figure 13.3: Structure of the Trigger-System

In code, it looks like this:

```
Trigger.in_state(
        <State>
).once(
        lambda: <Conditions>,
).execute(
        <Actions>
)
```

As the state and condition are optional, a trigger can also be written as shown in the following.

```
Trigger.immediately().execute(
        <Actions>
)
```

This is used, when an action should run at the start of a scenario, or in combination with the state-check to execute actions when entering a state.

The state refers to the current state of a state machine. (see Section 13.4) It is useful to differentiate, in which stage of the test-scenario a Trigger should be checked. For example, when overtaking, the minimum distance between the cars should not remain the same throughout the entire test.

This way, it is decoupled from TurtleCar-Core's internal state. If TurtleCar-Core behaves in an unexpected way - e. g. by outputting the wrong velocity values - the test-outcome is still based on the observed behavior of the vehicle.

The state is not constantly being checked. Only when a state machine

changes its state, will all triggers state-checks be evaluated. Triggers with a matching state-check will be placed in circulation and their condition will be regularly checked, while Triggers, where the state-check didn't match will not be checked, to not waste performance.

The condition is a free-form python lambda expression, which has to return something truthy, optimally a boolean. This expression is evaluated, every time the Test-Platform receives updated positional data for a simulation object from Gazebo. Whether a condition is met, is based solely on the data given to the Trigger-System by Gazebo.

Conditions can be formulated with the helper-methods provided by TurtleCar-Test. They allow checking positional and temporal constraints, as well as other aspects of the robot under test. That includes but is not limited to:

- the absolute position,
- the velocity,
- the steering angle,
- the activated driving functions,
- the time elapsed between certain points in the test procedure,
- the distance to other objects or actors, and
- the position in relation to other objects or actors.

Should the condition be met, the list of actions associated with this Trigger will be executed, and the Trigger will be removed from circulation. A trigger can never execute twice. An Action can be one of the following:

1. a positive or negative test outcome, followed by the end of the test
2. changing the current state of a state machine
3. starting, stopping, or resetting a Timer
4. changing the currently active driving-function
5. changing how the simulated driver acts

13.3 Timers

One or more timers can be specified in the test scenario specification file. It can be started, stopped, and reset as one of the actions of a Trigger. The current amount of elapsed time can also be used as a condition in a Trigger.

A timer can be useful to measure how much time has passed in-between different points of a test. It may be used to measure the elapsed time that a car takes to overtake another car, or as an abort, if a test fails for an unexpected reason and no other trigger is hit, that could end the test.

A timer can also be named. Its name will then be printed in the logs for enhanced debuggability.

This is how a timer is defined and used as a condition and in an action:

```
test_timer = Timer(name="TestTimer")

Trigger.once(
        lambda: test_timer.current_time() > 1.0,
).execute(
        test_timer.start(),
        test_timer.stop(),
        test_timer.reset(),
)
```

13.4 State Machine

State machines can be used to separate different parts of a test scenario. Zero or more can be used within the same test scenario specification file. They can be used to limit which conditions are tested for in different parts of the test. A state machine requires a list of states and a starting state as parameters and optionally a name. The name will then be printed in the logs for enhanced debuggability, whenever the state machine changes its state. Every Trigger can include an optional check for the current state of a state machine. Such a Trigger's condition will not be checked until the expected state is reached. It will also cease to be checked, when the state changes away from the expected state.

As part of an action, a state machine can be put into the next state or set to a specific state.

In code, the definition and usage of a state machine looks like this:

```
test_sm = StateMachine(name="TestSM", start_state="a",
        states=["b", "c"])

Trigger.in_state(
        test_sm.in_state("a"),
).immediately().execute(
        test_sm.set_state("b"),
        # or
        test_sm.next_state(),
)
```

13.4.1 Scenario

A `Scenario` is always the start of any test-scenario-specification. Its Constructor allows defining the world that the Gazebo simulator should load. It also defines a so-called Area-File path. An Area-File contains a list of named polygons. These named polygons can then be retrieved and used for checking against the position of the robot. For example, the current implementation of the simulated world contains three lanes. Their coordinates are stored in this file and allow easy and readable checks, whether a robot is inside a lane or not. The syntax for defining areas is as follows:

```
<area name 1>:<x1>,<y1>;<x2>,<y2>;<x3>,<y3>;...
<area name 2>:<x1>,<y1>;<x2>,<y2>;<x3>,<y3>;...
```

If there is no Area-File for a Gazebo Map, `None` still has to be explicitly specified for this parameter. The reason for this is, to prevent the user from forgetting to set this parameter.

There are more truly optional parameters that follow, defining the ROS domain ID, the path from which Gazebo should load its models, and the name of the point to which all other points are relative.

An example definition of a `Scenario` follows:

```
scenario = Scenario(
    gazebo_map_path="gazebo_data/empty_road.world",
    areas_map_path="gazebo_data/empty_road.areas",
    # optional
    ros_domain_id=0,
    gazebo_models_path="./gazebo_data",
    base_footprint_name="base_footprint",
)
```

13.4.2 Robot

Placing a Robot into the simulation is as simple as calling the Constructor. It requires a few parameters, which are important for TurtleCar-Test to associate the correct simulation-object with this Robot.

robot_id This parameter describes the unique name of this simulation entity. All references to his object within the Test-Platform use this name to refer to this instance

gazebo_model_path This parameter needs to point to a `.sdf` file, containing an XML-description of an object that Gazebo understands and can spawn. Within it, the text `ODOMETRY_FRAME_HERE` will be replaced by this robot's `robot_id`. This is necessary, because the robot's position information is tagged with the robot's odometry-frame name. Since this ID is part of the model file, the placeholder is needed.

robot_namespace This parameter should ideally match the `robot_id`, but does not have to. It assigns the robot its own namespace within ROS, which prevents multiple robots from interfering with each other.

turtlecar_command This parameter is the command executed to launch TurtleCar-Core. Technically any other program can also be started like this. The specified programm will be launched at the start of each test and stopped at the end. If the process does not yield to a `SIGTERM` after a timeout, then a `SIGKILL` will be sent. The command should also include the corresponding robot's namespace via the parameter `-ros-args -r __ns:=/<namespace>`. Followed by that are the spawn coordinates and rotation (or heading) of the bot. These have to be floating point numbers and are given in real-world meters and degrees of rotation.

outline Since the Test-Platform itself cannot read Gazebo model files and only receives the center position and rotation of each robot, it has to be given a rough outline of the robot to perform collision detection. Simple Rectangles can be specified using the `RectangleOutline`. More complex outlines require a list of vertices be given to the generalized `ObjectOutline` class.

What follows are optional parameters.

starting_speed This parameter tells TurtleCar-Core to start its simulation with the vehicle already rolling. This eliminates acceleration from the testing time. This feature is currently not fully implemented. Its value can be given as unscaled $^{m}/_{s}$.

target_speed and steering_angle These parameters are settings for the SimulatedDriver to assume. These values can be overwritten from a Trigger. The speed is given in unscaled $^{m}/_{s}$ and the steering angle is given in radians. Positive values steer to the left and negative numbers to the right. Zero is straight forwards.

enabled_driving_functions This parameter tells TurtleCar-Core which driving-functions to enable. In addition to that, the option `acc_target_speed` lets you set the speed, at which the ACC should keep the vehicle, when active. The driving-function `GUI` is a bit special. Its function is to enable or disable the TurtleCar-Core graphical debugging utility (usually just called Visualizer).

An example with all these values filled out is found below:

```
ego_bot = Robot(
    robot_id="burger_0",
    gazebo_model_path="gazebo_data/template_burger.sdf",
    robot_namespace="burger_0",
    turtlecar_command="python3 ../turtlecar/turtlecar/main.py"
           + "--disable_gui --mode simulation --ros-args" +
           "-r __ns:=/burger_0",
    x=0.0,
    y=0.0,
    z=0.0,
    rotation=0.0,
    outline=RectangleOutline(length=0.10, width=0.187),
    # optional below
    starting_speed=0.0,
    target_speed=0.1,
    steering_angle=0.0,
    enabled_driving_functions=[DrivingFunction.LKA],
)
```

13.4.3 Obstacles

To add an obstacle to any world, it is sufficient to call its constructor.

There, it is required to specify its position, rotation, the shape of the object-collision, and the path from which the gazebo model shall be loaded. This model is then spawned at the given place.

The following example creates a box that has ArUco-markers on each of its four corners. It is also a physics object, so it can fall and can be pushed.

An example that spawns an obstacle one meter in the air and two meters in front of the bot is found below:

```
Obstacle(
        x=2.0,
        y=0.0,
        z=1.0,
        rotation=0.0,
        outline=RectangleOutline(length=0.10, width=0.10),
        gazebo_model_path="gazebo_data/box_obstacle.sdf",
)
```

13.4.4 Simulated Driver

The Simulated Driver has access to the same controls that a human behind the steering wheel of a car has. It can control acceleration and steering controls.

As part of the Test-Platform, this module takes care of keeping a set speed and direction when no driving-function is active. It can also be used to test oversteering a lane-keeping assistant. It can be adjusted by the Trigger-System by way of a `DrivingFunctionChange` or a `SimulatedDriverSettingsChange`. These Objects cannot be directly created, but must instead be be created via the helper-methods of the `Robot` class. This ensures the association of these changes with a robot and a trigger, as these things can only be done as part of an Action. The following example shows the enabling of the lane-change-assistant and changing the settings of the simulated driver, such that it releases the steering wheel and holds the speed at 0.1 $^{m}/_{s}$ (no scaling is applied):

```
Trigger.immediately().execute(
        bot.change_driving_functions([DrivingFunction.LCA]),
        bot.change_sim_driver_settings(target_speed=0.1,
        steering_angle=None),
)
```

These parameters can also be given when spawning a `Robot`.

13.4.5 Gazebo integration

TurtleCar-Test relies heavily on Gazebo for providing the simulation environment and accurate positional data of each object within the simulation to evaluate conditions against. The Gazebo simulation system is started at the start of each test and shut down after its completion. During the test, it provides the positional data via the ROS topic `/tf`. The data provided by this topic is absolute and has no inaccuracies, unlike the simulated sensors that the simulated robots possess.

The Gazebo simulation only sends information about the current pose of any object. To obtain velocities, the positions are stored and the velocities are regularly calculated from the distance the bot traveled between the last two simulation steps. When the time between the steps is too small, this can result in wrongly calculating very high velocities. To avoid this problem, the velocity

is only recalculated when the robot has moved at least one millimeter or at least one millisecond has passed.

13.5 Implementation of TurtleCar-Test

The startup procedure is shown as an activity diagram in Figure 13.4. It begins by starting its own ROS node for publishing and subscribing to topics. This is used for the communication between robot, simulation, and TurtleCar-Test itself. It additionally starts the `TransformListener`, which specifically subscribes to the messages sent by Gazebo via the aforementioned `tf` topic containing information about the pose of the objects in the simulation, to obtain the position data of all simulated objects. Afterwards, the processes necessary to conduct the tests are started in the order given in the diagram. The visualization components are only started if the user sets the corresponding flag when starting TurtleCar-Test.

When the Test-Platform is started, the following things happen:

The `main.py` is called with one or more test-files or folders as parameters. The given locations are searched for valid python files. The following happens for each found test-specification:
The python file is dynamically imported into the Test-Platform itself. This creates all the objects, of which the constructors were called within the specification. After its execution, the specification is immediately unloaded again. The scenario that this testing specification described is however stored as a global variable in the `api.py`. From there, it is retrieved and the global variable is reset. What follows, is the actual buildup of a single test.

After startup, TurtleCar-Test loops through every Trigger and checks their state and the condition. If both are true, the associated action is executed. The action can range anywhere from a small adjustment in the SimulatedDriver to ending the test with a success or failure outcome. The state associated with a Trigger is only re-checked when said state changes, while the Trigger-condition is checked every simulation tick. Once a Trigger's action is executed, the Trigger will be discarded. If the action of a Trigger is a `TestOutcome`, then the Test-Platform will stop the Gazebo and TurtleCar-Core processes, and exit with a return message and a corresponding exit code.

When the user aborts the process by pressing Ctrl+C, the signal-handler in the aborting-module catches that signal. The handler interrupts the ongoing test and ensures an orderly cleanup and shutdown of the Test-Platform and its processes.

Programs that are started by TurtleCar-Test, like the Gazebo simulation or TurtleCar-Core, are launched as so-called subprocesses. Their outputs are then redirected to appear in the terminal and logfiles of TurtleCar-Test by the class `ParallelSubprocess`.

13.6 Future expansions

Currently, areas are only drawn, when the simulation is first started. While they are moved internally for distance and area calculations, they are never redrawn within Gazebo. This could be accomplished in multiple ways. One

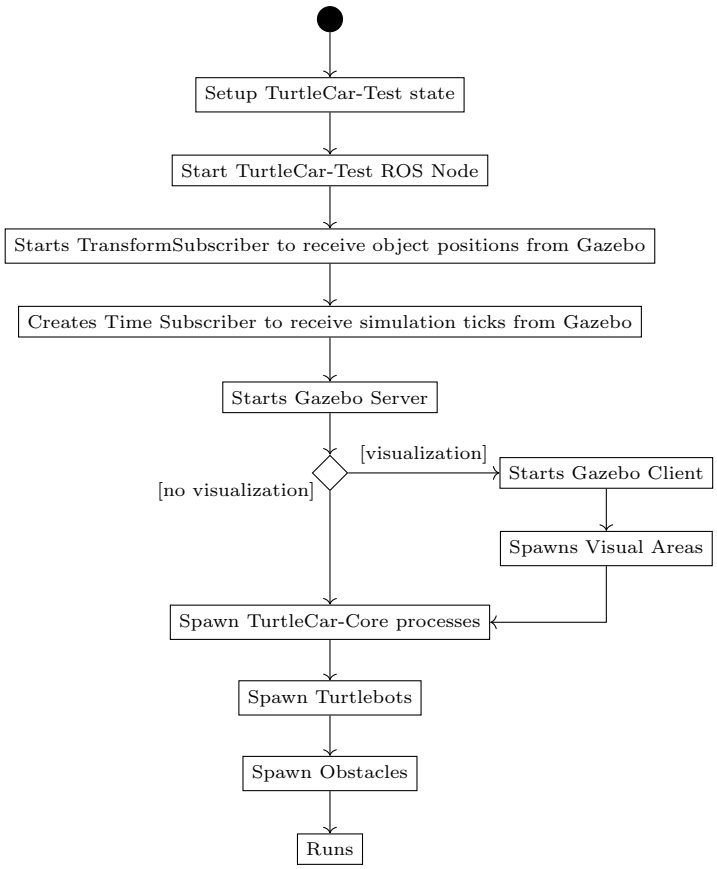


Figure 13.4: Activity diagram of the startup of TurtleCar-Test

option would be to write a Gazebo-Plugin that listens on a ROS-Channel for position updates and moves the areas without deleting them. A second option would be to employ RViz2 and only publish these areas on a topic that RViz can subscribe to, to display them. This option would also lend itself well to creating new areas by drawing them in RViz and publishing them to a topic that TurtleCar-Test can write to a file. The third and easiest option would be to repeatedly instruct Gazebo to remove a moved area and recreate it at the new position.

Since Traffic-Sequence-Charts are a well recognized standard for communicating traffic scenarios, it would be useful to either generate them from the currently used file-format and/or consume them as scenario definitions. Because they are made for human consumption, there is no easily parsable data-format available. For this reason, the project group has opted not to interface with Traffic-Sequence-Charts and leave their integration for future works.

Chapter 14

Architectural Concepts

In order to implement the various driving assistance function in such a way that they can be combined as would be the case in a real car, there is a need for an architecture that supports this. In this section, several architectural concepts are described that form the basis of implementing the individual driving functions. First, a structure is introduced that binds individual driving functions together to form the autonomy levels described in Section 1.1. Afterwards, the usage of MPC within this project is described, which is used to implement certain driving functions.

14.1 Autonomy Level Architecture

This section describes two use cases which are derived from the vision (see Section 1.1). The first use case combines the autonomy levels "no autonomy" and "partially automated", while the second organizes the driving functions for highly autonomous driving.

14.1.1 Manual Driving and Partial Autonomy

This section describes the usage of the first two autonomy levels, Manual driving and partial autonomy, since they are closely related and implemented in one consistent architecture. The general principle is that the vehicle always starts in manual driving mode. Afterwards the human driver has the ability to manually activate assistance functions or deactivate them again. The inputs that can be given by the driver are shown in Table 14.1. The table shows the input mappings for keyboard and gamepad. Additionally, the inputs can be given via ROS parameters for automatic testing.

The interplay of the manual and assisted driving functions can be modeled as a pipeline. The architecture is described in Figure 14.1. The supervisor module contains an ordered list of control modules for driving functions which can be turned on or off. Each iteration of the control loop, it first asks the manual control function (described in Section 16.1) for its input. Afterwards, each controller in the ordered list that is switched on is asked for its input. Each controller writes its output directly to the `Action` interface. The next controller therefore has the possibility of overriding or altering input from the

Table 14.1: The keyboard and gamepad inputs for a human driver.

Function	**Key**	**XBOX Gamepad**	ROS Parameter [Arguments]
Increase Velocity	W	Press Right Trigger	`user_input_target_speed`
Decrease Velocity	S	Release Right Trigger	`user_input_target_speed`
Steer Left	A	B	`user_input_steering_angle`
Steer Right	D	X	`user_input_steering_angle`
Toggle Lane Keeping	K	Y	`user_input_toggle_lka`
Initiate Lane Change Left	Q	not set	`user_input_set_lca [left]`
Initiate Lane Change Right	E	not set	`user_input_set_lca [right]`
Abort Lane Change	P	not set	`user_input_set_lca []`
Toggle Overtaking	O	not set	`user_input_toggle_ota`
Emergency Stop	Space	Left/Right Shoulder Button	No parameter, send a message of type Twist with all zero values to topic `\cmd_vel`

previous controller. Some controllers may contain a planner which writes its planned path to the `Plan` interface. This may also be overridden by a subsequent controller. To avoid confusion, only one controller should therefore be allowed to develop a plan. In the future, this may be replaced by a using global planner, which develops a plan for all driving functions or by integrating multiple driving functions into one unit. Therefore, the pipeline architecture described here is subject to change. Currently, the pipeline consists of the following controllers in this order:

1. Manual Control
2. Lane Keeping Assistant
3. Adaptive Cruise Control
4. Lane Change Assistant
5. Overtaking Assistant

These controllers are described in detail in Chapter 16.

14.1.2 Autonomous Driving

The driving functions described in Subsection 14.1.1 enable controlling the lateral and longitudinal movement of the car in specific situations. Within the

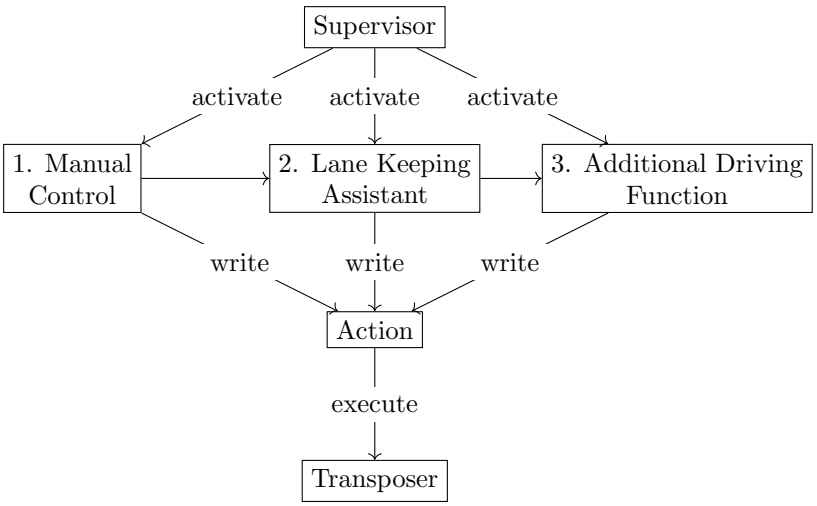


Figure 14.1: Interplay of supervisor and controllers for the individual driving functions.

individual driving functions' operational constraints, it is up to the driver to decide which driving functions should be enabled in which situations. For autonomous driving, the same driving functions as for assisted driving are used, but the decision process is automated. For this, the supervisor structure is extended by introducing components called „Interpreters". These interpret the information stored in the model in order to determine the situation the vehicle is currently in. The interpreters can also be combined with existing controllers, as they already interpret the environment in order to determine the correct control action. For example, an ACC always needs to check if there is an obstacle ahead to which to adapt the speed. Therefore, it can publish that information as an interpretation of its environment. The information gathered by the interpreters is used by the supervisors to decide which controllers to enable. This structure can be seen in Figure 14.2.

The supervisor consists of a state machine which is triggered by the „update" signal each controller cycle. When the „update" event is triggered, all interpreters are updated in order to present the current information to the supervisor. After this, the transition the state machine takes is determined by the state the supervisor is in and transition conditions based on the interpreter information. In order to implement the state machine, a framework for building state machines in python is used [51].

Using this concept, the following two different supervisor structures representing different driver characteristics are implemented. A detailed description of the interpreters used can be found in Chapter 15. Likewise, the driving functions are described in Chapter 16.

Passive Autonomous Driver

This autonomous driver always tries to stay in the current lane and match the current speed limit. If another slower car is in its lane, it slows down and matches the speed of that car. It never overtakes. A representation of the

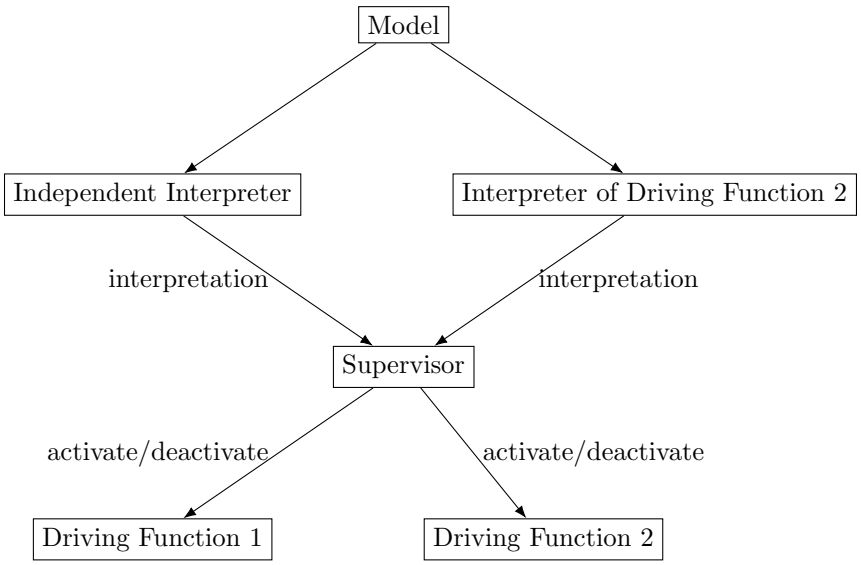


Figure 14.2: The components of the autonomous supervisor structure.

automaton describing the behaviour is shown in Figure 14.3.

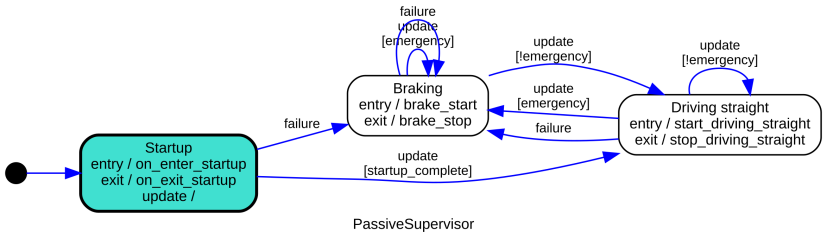


Figure 14.3: The state machine encoding the behaviour of the passive autonomous driver

The driver uses the LKA, the ACC and the Emergency Brake as driving functions to achieve this behaviour. In the diagram, this is denoted by the action „start driving straight“. To determine whether it should activate the emergency brake, it uses the Emergency Detector as interpreter which determines if there is an object close to the front of the car. The supervisor starts in a startup state, where it drives straight slowly for a few seconds in order to gather enough sensor information so that the EKF can robustly estimate the vehicle's position and the lanes are detected. In addition to the update transition, a special failure transition goes from each state to the braking state. This models the case that the driving function that is currently active malfunctions, in which case an exception is thrown in the code. The supervisor catches this exception and immediately sends a failure signal, which triggers a state change to the Braking state. The change happens immediately, ensuring that the malfunctioning driving function does not issue a wrong command. It is assumed that the braking function itself is not able to fail, as it only issues one static braking command

to the vehicle. In the next update step, the automaton will try to resume the Driving Straight state. If it fails again, it will immediately change back to the Braking state. Therefore, as long as a driving function malfunctions, no wrong commands will be issued.

Maximum Speed Driver

This autonomous driver also tries to stay in the current lane and match the current speed limit. However, if a slower car is ahead, the driver is allowed to overtake it.

This driver is an extension to the Passive Driver. It uses the LKA, the ACC, the Emergency Brake and the Overtaking Assistant (OTA) as driving functions. It uses an Overtaking Interpreter to determine whether it should overtake and the Emergency Detector to recognize the need for an emergency braking maneuver. Similar to the Passive Driver, an error in an active driving function results in a failure signal and the transition to the braking state. The automaton implementing this behaviour is shown in Figure 14.4.

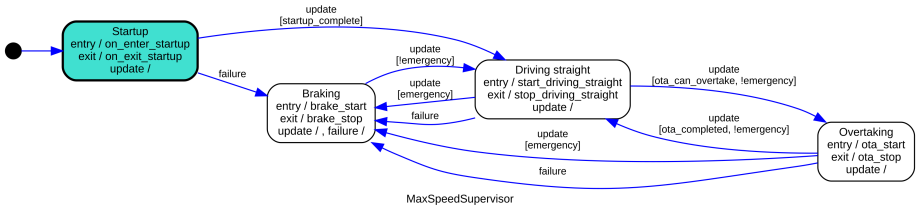


Figure 14.4: The state machine encoding the behaviour of the passive autonomous driver

In comparison to the Passiver Driver it has an additional overtaking state, which is entered as soon as the vehicle detects the possibility to overtake. It stays in this state until the OTA has finished the maneuver and the vehicle is back in its original lane.

14.2 Model Predictive Control

In this project group, the aim is to build driving functions which enable solving different scenarios. In order to achieve this in a way that consistently provides good results, a two-layered approach for the control strategy was chosen. A planner calculates a trajectory and boundary conditions, which are then used by a MPC to derive the best possible solution. This approach is heavily based on the work of Wunderli [97] and of Kröger [44], who used MPC to achieve optimal tracking of a given trajectory with a model racing car. The approach is almost directly transferable as the model it uses is the same bicycle model used in the project group, as defined in Section 5.1. This section aims to give an overview and guidelines on how new driving functions can be implemented by designing a problem consisting of a trajectory and boundary conditions which can then be solved by the MPC, resulting in optimal control signals. First, an overview over the MPC approach used in this project group is given. This also includes information on where the project group deviated from the approaches

described by Wunderli and Kröger. Afterwards, an architecture is described which incorporates the MPC and implementation guidelines are given.

14.2.1 The Model Predictive Control Algorithm

Wunderli describes how to model the trajectory tracking problem as a quadratic problem which can be solved by existing solvers. Kröger deviates from Wunderli's original approach in some ways. These deviations were incorporated in this project group. It is assumed that a trajectory is given which can be tracked. The trajectory has the form

$$\vec{x}_n(t) = \begin{bmatrix} v_n(t) & \psi_n(t) & x_n(t) & y_n(t) & a_n(t) & \omega_n(t) & \kappa_n(t) \end{bmatrix}^T$$

where for each point in time t the nominal state $x_n(t)$ is completely defined. In addition to the state variables used in the bicycle model, it also contains values for the nominal trajectory curvature κ_n as well as the nominal acceleration a_n and angular velocity $\omega_n = v_n \kappa_n$ as they are needed for subsequent calculations. Using this nominal trajectory and the dynamics of the bicycle model, a model of the error dynamics describing of the deviations between the nominal and the actual trajectory can be derived. This model is linearized around $\psi_e = 0$, $v_e = 0$ while approximating $\cos(v_e) = 1$ and $v_e \sin(v_e) = 0$. The linearized error dynamics are then described by

$$\dot{x}_e = \begin{pmatrix} \dot{v}_e \\ \dot{\psi}_e \\ \dot{x}_e \\ \dot{y}_e \end{pmatrix} = \begin{pmatrix} a - a_n \\ \omega - \omega_n \\ v_e + \omega_n y_e \\ v_n \psi_e - \omega_n x_e \end{pmatrix}$$

with $\omega = \frac{v}{L}\delta$. When discretized with time constant T, the error dynamics are defined by

$$x_{e,k+1} = A_k \cdot x_{e,k} + B \cdot \begin{pmatrix} a_k \\ \omega_k \end{pmatrix} - C_k \tag{14.1}$$

where

$$A_k = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ T & 0 & 1 & T\omega_{n,k} \\ 0 & Tv_{n,k} & -T\omega_{n,k} & 1 \end{pmatrix},\ B_k = \begin{pmatrix} T & 0 \\ 0 & T \\ 0 & 0 \\ 0 & 0 \end{pmatrix},\ C_k = \begin{pmatrix} Ta_{n,k} \\ T\omega_{n,k} \\ 0 \\ 0 \end{pmatrix} \tag{14.2}$$

In order to obtain the optimal control signals which minimize the error between the actual and nominal state for N time steps, the following optimization problem is defined:

$$\min_u \sum_{k=0}^{N} \begin{pmatrix} v_{e,k} \\ \psi_{e,k} \\ x_{e,k} \\ y_{e,k} \end{pmatrix}^T Q_k \begin{pmatrix} v_{e,k} \\ \psi_{e,k} \\ x_{e,k} \\ y_{e,k} \end{pmatrix} + \sum_{k=0}^{N-1} \begin{pmatrix} a_k \\ \omega_k \end{pmatrix}^T R_k \begin{pmatrix} a_k \\ \omega_k \end{pmatrix} + w_\epsilon \epsilon \tag{14.3}$$

$Q_k \in \mathbb{R}^{4\times4}$ is the weighing matrix for the error states and $R_k \in \mathbb{R}^{2\times2}$ is the weighting matrix for the inputs. They contain constants on their diagonal which

weigh the respective input and state variables. ϵ and its weighting factor w_ϵ represent the cost in case of the vehicle exceeding lateral constraints. This is described further in the constraints definition below. The weighting factors can change each step, but in the simplest case they stay the same. Note that the weights for the velocity and the position should not be chosen to be the same, since the focus for the cost function needs to be set on either one. This optimization problem needs to be subject to the following constraints:

Error Dynamics The optimization must be conducted along the dynamics of the system, so per Equation 14.1 and Equation 14.2 the following condition must be met:

$$\forall k \in [0, N-1] : \begin{pmatrix} v_{e,k+1} \\ \psi_{e,k+1} \\ x_{e,k+1} \\ y_{e,k+1} \end{pmatrix} = \begin{pmatrix} v_{e,k} + T \cdot (a_k - a_{n,k}) \\ \psi_{e,k} + T \cdot (\omega_k - \omega_{n,k}) \\ x_{e,k} + T \cdot (v_{e,k} + \omega_{n,k} \cdot y_{e,k}) \\ y_{e,k} + T \cdot (v_{n,k} \cdot \psi_{e,k} - \omega_{n,k} \cdot x_{e,k}) \end{pmatrix} \tag{14.4}$$

Initial Error State The current state of the system needs to be given to the MPC for it to be able to generate a suitable control signal to steer the error to zero. The initial error state is therefore fixed as follows:

$$\begin{aligned} v_{e,0} &= v - v_{n,0} \\ \psi_{e,0} &= \psi - \psi_{n,0} \\ X_{e,0} &= (X - X_{n,0}) \cdot \cos(\psi_{n,0}) + (Y - Y_{n,0}) \cdot \sin(\psi_{n,0}) \\ Y_{e,0} &= (Y - Y_{n,0}) \cdot \cos(\psi_{n,0}) + (X - X_{n,0}) \cdot \sin(\psi_{n,0}) \end{aligned} \tag{14.5}$$

Maximum Steering Angle The steering angle of a car is limited. Since not the steering angle itself, but the input $\omega = \frac{v}{L}\delta$ is used, the constraint on the steering angle is

$$\forall k \in [0, N-1] : \frac{v}{L}\delta_{min} \leq \omega \leq \frac{v}{L}\delta_{max} \tag{14.6}$$

Maximum Longitudinal Acceleration Because of restrictions on the vehicle's abilities, the acceleration must be constrained:

$$\forall k \in [0, N-1] : a_{min} \leq a_k \leq a_{max} \tag{14.7}$$

Maximum Lateral Acceleration For passenger convenience, the lateral acceleration $q = v\omega$ should not exceed a certain value. Because $v\omega$ can not be expressed in a linear MPC, it is assumed that $v \approx v_n$. Therefore the constraint is formulated as

$$\forall k \in [0, N-1] : v_n \omega_k \leq q_{max} \tag{14.8}$$

.

Position Constraints The space on which the car may drive is limited by several factors, i. e. the lane and road boundaries or obstacles. For now only lateral position constraints based on y are considered, but in principle obstacles could also be modeled by using constraints on x. It may happen that the vehicle is just at the boundary of a forbidden state and may at some points in time not be able to avoid that state. This would result in an infeasible problem and an inability to provide control signals via MPC even though it would have produced control signals that would drive the vehicle away from the forbidden state. Therefore, instead of imposing hard constraints on the position, this constraint is used to define a relation to a factor ϵ which gets higher the more the constraint is violated. Together with a very high weighting factor w_ϵ, this drives the const for violating the constraint so high that the MPC will try to keep the vehicle out of the forbidden positions.

$$\forall k \in [0, N] : y_{e,k,min} - \epsilon \leq y_{e,k} \leq y_{e,k,max} + \epsilon \tag{14.9}$$

14.2.2 Implementation

The MPC is implemented by building the optimization problem from a given trajectory and given position constraints and solving it using a suitable solver for quadratic problems. In this project group, the Gurobi solver is used as it has been successfully used for similar problems previously [44], [9]. In contrast to Matlab, which also provides functions for solving optimization problems [55], Gurobi can be used via a Python interface [28], which makes it possible to quickly integrate it into the existing architecture.

Usage Driving functions need to provide a trajectory and position constraints in a standardized format to the component building the optimization. The architecture is depicted in Figure 14.5. It shows how a driving function can obtain optimal control signals through a standardized interface by providing a trajectory and constraints. From these, a so-called Problem instance is constructed in which these informations are modeled as a quadratic problem. The Problem instance communicates with the Gurobi solver to generate a series of optimal control signals, the first of which is handed back to the driving function.

In order to speed up the development of new driving functions, the MPCProblem class has been implemented in a way that enables quick definition of trajectories and constraints and obtaining optimal control signals which drive the car according to these specifications.

An example of a driving function that uses the MPC to obtain control signals is described in short in the following, using natural language. The driving function could be implemented like this quite easily in TurtleCar, since TurtleCar exposes several software components that are of help. Similar driving functions based on the MPC can be found in TurtleCar.

First, the time constant is set: $T = 0.25$. Additionally, constraints are created. The acceleration should be between 0.0 and 0.2 $\mathrm{m/s^2}$. TurtleCar exposes a simple software interface for creating constraints.

Now, a trajectory comprised of ten points, discretized by T, can be calculated. The ten points are defined by individual states, which contain the current velocity at the state and some constraints, in this case, the acceleration con-

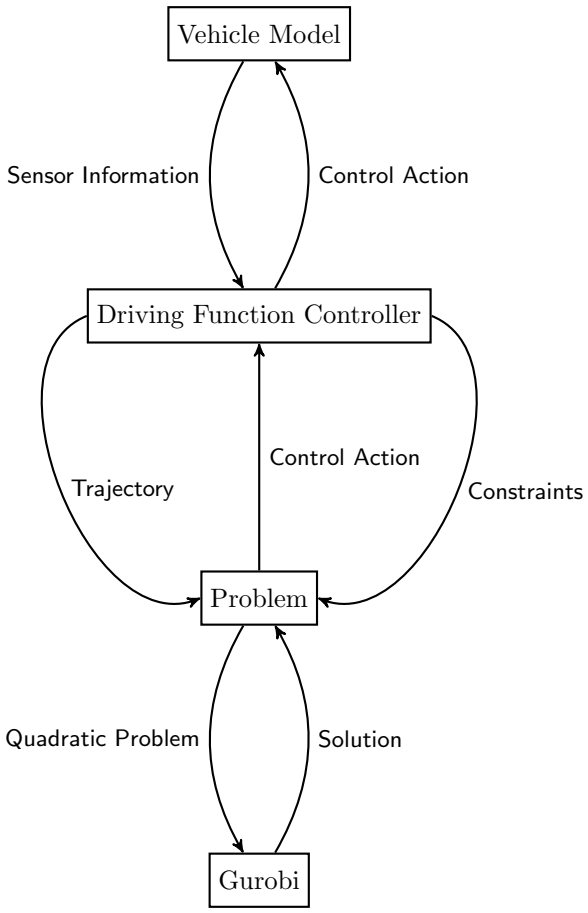


Figure 14.5: The architecture for intergrating MPC in TurtleCar by providing a problem builder as interface between driving function and optimizer.

straint mentioned above. This example trajectory spans 2.5 s in total, because of T.

Now that the trajectory is defined and constraints are given, a problem instance for the MPC can be created. It exposes control signals, which can be used to alter the user input model values of TurtleCar, resulting in essentially applying them to the current driving scenario.

Combined, this driving function implements a cruise control which keeps the car at a speed of 36 $^{\mathrm{m}}/_{\mathrm{s}}$. The acceleration is constrained to a maximum of 2 $^{\mathrm{m}}/_{\mathrm{s}^2}$. For this, the list of states is defined from which a trajectory is generated.

Generating Trajectories In order to be able to easily create trajectories, methods to generate full trajectories from limited information are implemented as part of the MPC toolset in TurtleCar. A trajectory can be generated from a list of states where each state contains one of the following combinations of state information:

- x and y

- x, y and v
- x, y, v and a
- v and ψ

These methods use the given state information to estimate the remaining state information. The given state information may be inconsistent in the sense that its reference states do not work together. E. g. , the difference in position between two states over the timestep may not be consistent with the velocity in these states. This is permitted, as the MPC does not need a fully consistent trajectory in order to minimize the error.

To convert the list of states to a trajectory, it is required to define a constant time T between the states. In order to follow a trajectory without needing to re-calculate it from the current state of the bot, it is necessary to find the trajectory index that represents the next state depending on the current state of the car. The trajectory is implemented in such a way that, when iterating over it, it starts the iteration at the state following the current state of the vehicle. For this, the two states in the trajectory with the least distance to the current car state in terms of x and y are found and the state with the higher index is selected as the next state.

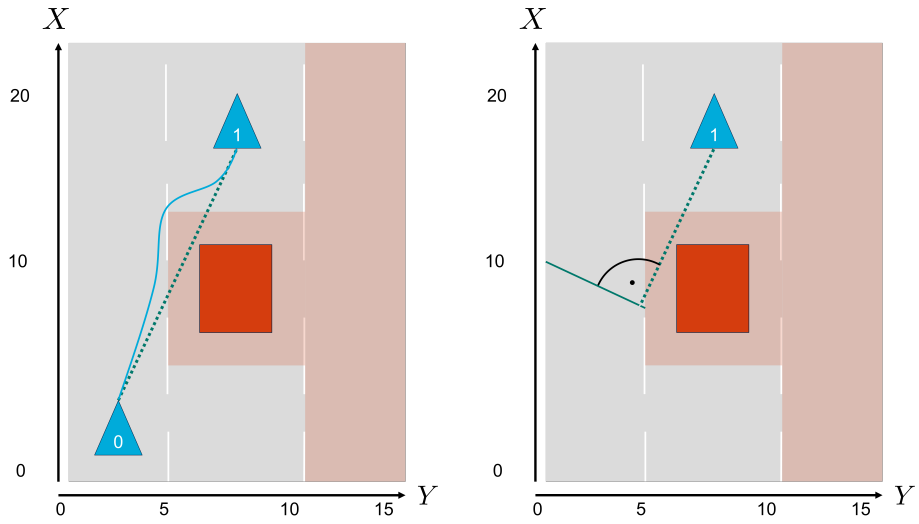
It may be necessary to re-plan the trajectory, e. g. , because the situation the car is in changed or to compensate sensor inaccuracies building up over time. To be able to achieve this, states can be labeled with integers. These labels can be used to identify sections of a trajectory. The sections can be labeled arbitrarily by the planner, but usually they should be labeled in ascending order as this enables comparing whether the current label is higher or lower than other sections. From this comparison, the planner can determine if it still has to plan that section or can discard it. A trajectory can tell how many steps are left when counting from the current vehicle state until the label of the state changes. When given the old trajectory with labeled states and the current state of the vehicle, a trajectory planner can utilize this to determine how many steps it has left to plan in the current trajectory section. An application of this is shown in Section 16.4.

Defining Constraints Each state can be assigned constraints. In the example above, all states are assigned the same constraint. The trajectory is used together with constraints for the initial inputs to build an instance of MPCProblem, from which the next control signals are obtained. Internally, the optimization problem is solved using Gurobi for the next N time steps and the first control signals are returned.

Additionally to the constraint on acceleration shown above, the Y position can be constrained too. This way a whole lane can be constrained, meaning the vehicle is not allowed to drive there. This applies even if the trajectory leads through the forbidden lane.

Given the trajectory the required deviation from the trajectory will be calculated to evade the forbidden lane. Per default, the calculation is implemented in a way, that makes sure, the vehicle always stays on the road. In other words, every area outside the street counts as forbidden.

Such a way of constraining lanes can be used for obstacle avoidance. If an obstacle is positioned on a lane, this lane can be considered forbidden and the



(a) Obstacle evasion using forbidden lanes

(b) Calculation of deviation from trajectory for obstacle evasion

Figure 14.6: Obstacle evasion by defining forbidden lanes

vehicle would evade it. This is depicted in Figure 14.6a. The planned trajectory goes through a forbidden area, where an obstacle is situated, but by adding this lane to the constraints the vehicle will not drive through it. Instead it will try to adhere to the trajectory as closely as possible while still not entering the forbidden area and also obeying the other constraints.

To achieve this behavior the allowed deviation from the trajectory in lateral direction is calculated. To do this, for each trajectory's state a line orthogonal to the heading is created as exemplarily seen in Figure 14.6b. Next, the intersection in both directions to a forbidden area is calculated to compute the relevant distances $y_{e,k,min}$ and $y_{e,k,max}$ from Equation 14.9. Additionally, the width of the vehicle is incorporated into this calculation by removing half of the vehicle's width from the allowed deviation on either side.

In the example, the allowed deviation from the trajectory ranges from a negative number to a positive one. The vehicle can drive into either direction while staying in the specified range. If, however, the vehicle was already in a forbidden area, the algebraic sign before the two distances are the same and indicate in which direction the vehicle evades to. A negative sign indicates an evasion to the left, a positive sign an evasion to the right.

The described approach for obstacle avoidance works conceptually. However, despite the right calculation of the allowed deviation, ϵ does not increase which it should when the constraint is violated. The vehicle drives into the forbidden lane anyway, not changing its course specified by the constraint. This was tested in different scenarios in the simulation. Since the driving function of evading obstacles can also be implemented without forbidden lanes and the project reaches its end, this issue is not resolved and another way of implementation was chosen as described in Chapter 16. To revisit this in the future, the described implementation is available but currently inactive.

Chapter 15

Situational Awareness

Especially the autonomous driving functions require interpreting the data collected within the vehicle's model in order to gain an understanding of the situation. As this may be very context-specific and require complex processing steps, this interpretation is outsourced into task-specific components called Interpreters. They are mainly used by autonomous supervisors to decide which driving functions should be activated depending on the situation, as described in Subsection 14.1.2. They are executed each time the supervisor is called before it makes its decision, so that the actions performed by the supervisor are always based on the latest interpretation of the situation. Some of these interpreters share logic with driving functions that are closely related, and therefore were integrated together with them in one module. This chapter describes the interpreters that are not directly integrated with other driving functions. The remaining interpreters are described together with their driving functions in Chapter 16.

15.1 Emergency Detector

This detector is used as a safeguard to prevent or at least mitigate crashes in the case that other driving functions have driven the vehicle in a state that is unsafe. It uses the raw lidar information of the model and checks if the minimum of the lidar rays facing in the direction set by the steering wheel measured a distance $d_e < \max(d_{min}, v/2)$ where d_{min} is a velocity-independent minimum safety distance and v is the current speed of the vehicle in $^{\mathrm{km}}/_{\mathrm{h}}$. If that condition is met, the emergency detector returns the EMERGENCY interpretation to the caller. It was experimentally determined that it is sufficient to use the nearest 3 lidar rays in steering direction as a basis, as using a wider angle would lead to falsely interpreting situations as emergencies. Otherwise, it returns an empty set of interpretations.

15.2 Lane Change Safety

During a lane change, it is possible for obstacles to disrupt the vehicle's plans. Therefore, in this subsection, the strategy and implementation for avoiding obstacles during a lane change will be described. First, a strategy is defined which

formalizes rules on how to avoid obstacles during lane changes. Second, the strategy is implemented as an Interpreter usable in the architecture defined in Chapter 14.

The idea is that the vehicle must not violate safety distances of other cars or its own during a lane change. The distances to obstacles from the position of the vehicle during the lane change are calculated, and compared to the obtained safety distances. If a safety distance is violated, the lane change will be blocked.

The car should not collide with an obstacle before the lane change, during or after it. It does not matter whether the obstacle is static or moving, a collision should be avoided.

One could argue that this is the case with mostly all vehicle operations - an obstacle should never be collided with. However, this general case changes dramatically when the driver signals a lane change. Now, several more considerations have to be made. It has to be determined if the vehicle would collide with a vehicle up front after the lane change or if there are vehicles behind that are significantly faster and would have no time to slow down appropriately. See Figure 15.1 for an overview of the problem.

15.2.1 Strategy

Lee et al.define strategies for avoiding collisions during lane changes, formalize the problem and implement a supervisor for the LCA. Fortunately, the TurtleCar software already provides a supervisor which can be used to prevent a lane change if it is deemed unsafe. This decision is implemented as an Interpreter module (Lane Change Assistant Interpreter (LCAI)). Based on Lee et al., an algorithmic strategy is designed which takes safety distances and current obstacle distances into account [47]. The strategy for solving the problem is formulated for TurtleCar in the following:

During a lane change, obstacles must be avoided. If one of the following conditions is met, the lane change maneuver must be blocked.

1. There exists an obstacle on the lane to change to, it is further behind the ego vehicle on its lane, and the obstacles relative speed to the ego vehicle is significant and positive. The relative speed of the obstacle is significant if it is bigger than $30\,^{\mathrm{km}}/_{\mathrm{h}}$. This value has been defined experimentally (see Figure 15.2).

2. There exists an obstacle on the lane to change to, it is further behind the ego vehicle on its lane, and the ego vehicle would violate the obstacles safety distance to the front or its own safety distance but to the back (see Figure 15.3).

3. There exists an obstacle on the lane to change to, it is in front of the ego vehicle on its lane, and the ego vehicle would violate its own safety distance to the front (see Figure 15.4).

The first rule exists to not slow down other obstacles immensely. When the vehicle does not violate safety distances, it could still mean that the obstacle would have to slow down immensely (i. e. , braking) in order to resolve the situation. This should be avoided, since a real driver also would not change onto a lane where a very fast obstacle relative to his own speed is approaching.

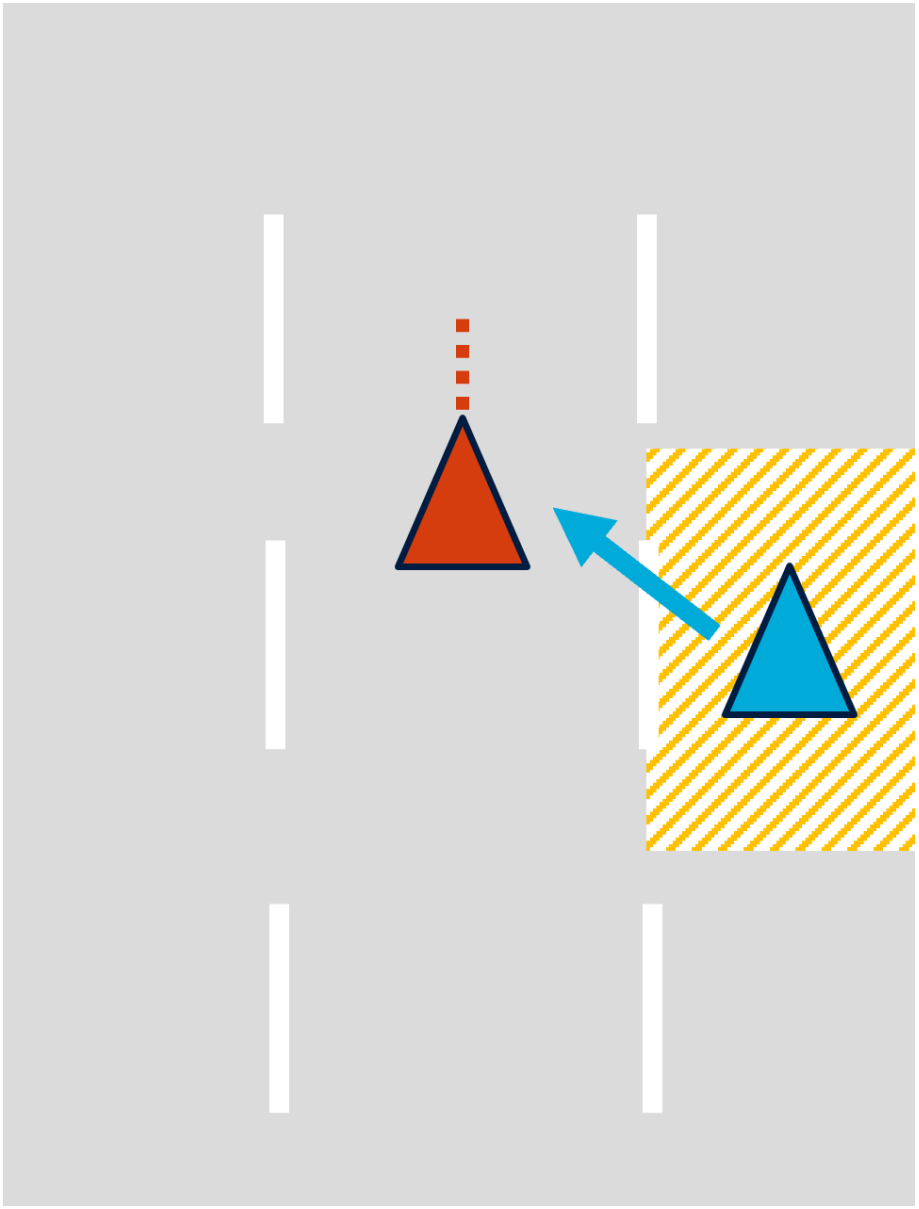


Figure 15.1: The general case of the problem

The second rule makes sure that obstacles behind the vehicle stay outside the safety distance. Both safety distances are applied, because the vehicle's speed is most likely to be more accurate than those of the obstacles. The third rule makes sure the vehicle keeps the safety distance to the front after changing the lane. Implicitly, these rules also make sure that the vehicle does not collide with obstacles directly beside him during the lane change, as their distance to the vehicle will always be smaller than a given safety distance based on vehicle speed. A situation where none of these conditions is met and the lane change

is therefore allowed is shown in Figure 15.5.

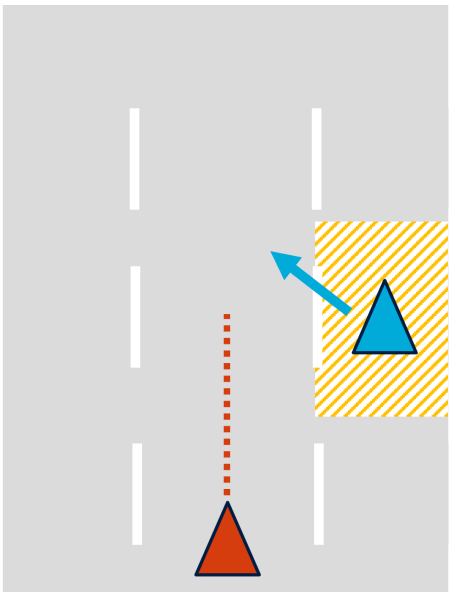


Figure 15.2: Fast approaching obstacle

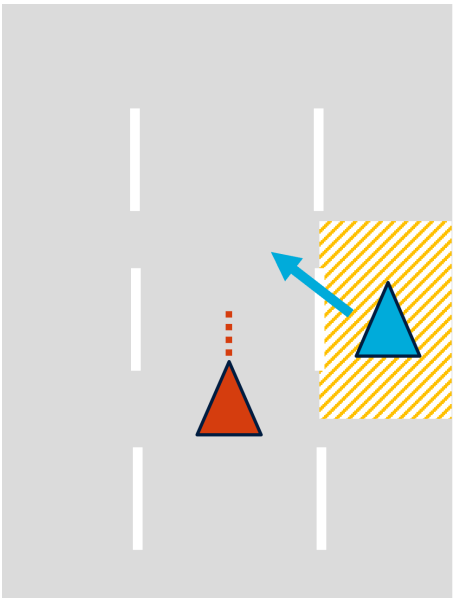


Figure 15.3: Obstacle behind

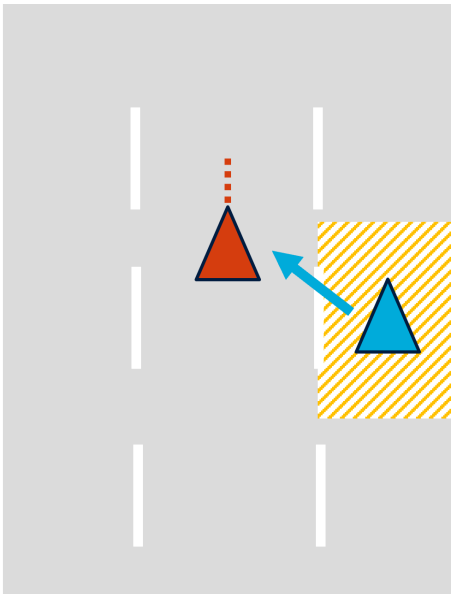


Figure 15.4: Obstacle in front

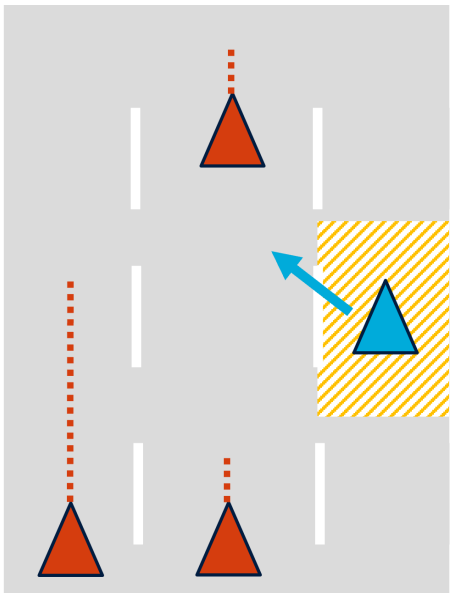


Figure 15.5: Allowed Lane Change

15.2.2 Implementation of the Obstacle Avoidance Strategy

In order to implement the strategy, the correct distances and safety measures have to be calculated in each update step and compared to the rules defined

above. The supervisor distributes the update steps, so that the LCAI only has to implement one base method which is called periodically by the supervisor. See Section 14.2 for more details on the supervisor, driving functions and controllers.

The calculation in each update step is using vector calculations and the path planner from Chapter 11. The following information about obstacles is present in the TurtleCar model, see Chapter 10 for details.

1. Position
2. Heading
3. Velocity
4. Relative velocity to ego vehicle

The following steps are taken to do the calculations.

1. **Calculating the projected position**
   The projected position, called $\vec{pos}_{tb,proj}$, is the position of the vehicle after having performed the lane change. It is calculated by using the median path planner also used in other driving functions, see Chapter 11 for details. The planner is instructed to plan a path in the middle of the lane to change to, starting from the closest point to the vehicle's current position. $\vec{pos}_{tb,proj}$ is extracted by using the first point of the planned path. This will be the closest middle point on the target lane.

2. **Identifying all relevant obstacles**
   The list of all obstacles is filtered to contain only obstacles that are located in the lane to change to. Obstacles that are not located in the lane to change to will not impose a problem for the vehicle. This is under the assumption that they don't interfere with the lane change process drastically. Rogue actors are not considered.

3. **Identifying the closest obstacle behind and the closest obstacle up front**
   From the list of relevant obstacles two specific obstacles are retrieved: the closest up front, and the closest to the back. This is done in a two-step process:

   (a) Separate the list into two lists, one containing all obstacles behind the vehicle and one containing all obstacles in front of it. This is done by calculating the dot product of the unit vector based on the vehicle's heading and the vector from the vehicle to the obstacle. If the dot product of these vectors is positive, both vectors point in the same direction, which means the vehicle is looking towards the obstacle - i. e. , the obstacle is in front of the obstacle. If it is negative, the obstacle is behind the vehicle. Zero is interpreted as the obstacle being in front of the vehicle.

   (b) These two lists are then filtered to find the obstacle with the least distance to the vehicle's position. This results in two obstacles: one closest behind and one closest in front.

4. **Calculating the distances to the obstacle up front and obstacle behind**
   $distance_{pos,proj,front}$ is the distance of the obstacle up front to $\vec{pos}_{tb,proj}$.

$$distance_{pos,proj,front} = ||\vec{pos}_{obst} - \vec{pos}_{tb,proj}||$$

   $distance_{pos,proj,back}$ is calculated like this as well.

5. **Calculating safety distances**
   $safety_{tb}$ and $safety_{obst}$ are calculated based on $||\vec{vel}_{obst}||$ and $||\vec{vel}_{obst}||$, respectively. The safety distances are the velocities converted to $^{km}/_{h}$ and then halved.

$$safety_{tb} = ||\vec{vel}_{obst}|| \cdot 3.6/2$$

6. **Deciding whether to block the lane change**
   After having now calculated all relevant parts, the strategy defined in Subsection 15.2.1 is followed. If a violation according to the strategy occurs, the lane change is deemed unsafe. This information is returned to the supervisor.

15.2.3 Testing

The scenarios described in the following define a way to test the functionality of the lane change interpreter. They are the basis for extensive parameterized unit tests included in the TurtleCar source code. Each scenario includes a description, a graphical overview and steps to replicate it.

Scenario 1: No obstacle

Description: This scenario is the situation where the LCAI is active, but no relevant obstacle is on the target lane. This aims to test whether the LCAI performs invalid blocking actions. It is shown in Figure 15.6.
How to replicate this scenario:

1. Initialize the simulation environment on a straight road with one vehicle.
2. Accelerate the vehicle.
3. Initiate a lane change to a new lane.
4. Monitor the LCA state. The action should be allowed and the lane change completed.

Scenario 2: Obstacle behind and inside safety distance

Description: This scenario is the situation where the LCAI is active, and there is an obstacle inside the vehicle's safety distance to the back on the target lane. It is shown in Figure 15.7. This aims to test whether the LCAI performs valid blocking actions regarding back obstacles.
How to replicate this scenario:

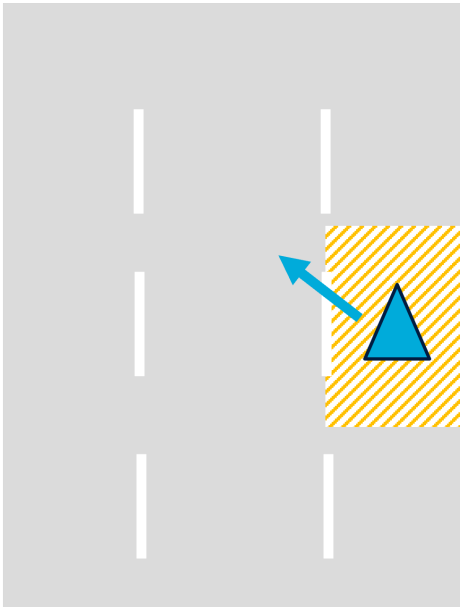


Figure 15.6: Allowed Lane Change

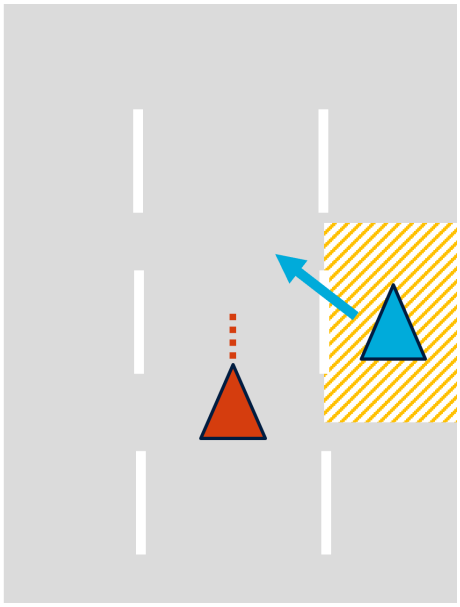


Figure 15.7: Approaching Vehicle within safety distance

1. Initialize the simulation environment on a straight road with two vehicles side by side.

2. Move the vehicle on the target lane slightly to the back of the ego vehicle that will perform the lane change.

3. Accelerate both vehicles to the same high speed in order to create a high safety distance.

4. Initiate the lane change to the lane where the vehicle is behind.

5. Monitor the lane change of the ego vehicle. The action should be blocked.

Scenario 3: Obstacle in front and inside safety distance

Description: In this scenario, which can be seen in Figure 15.8, there is an obstacle inside the vehicle's safety distance to the front on the target lane. This aims to test whether the lane change is prevented in response to frontal obstacles.

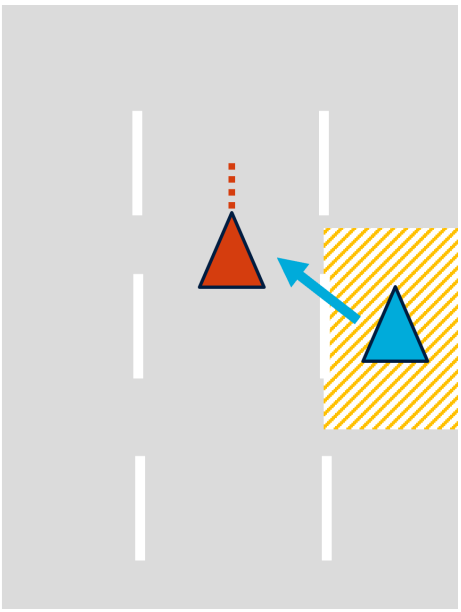


Figure 15.8: Vehicle within safety distance in new lane

How to replicate this scenario:

1. Initialize the simulation environment on a straight road with two vehicles side by side.

2. Move the vehicle on the target lane slightly to the front of the ego vehicle that will perform the lane change.

3. Accelerate both vehicles to the same high speed in order to create a high safety distance.

4. Initiate the lane change to the lane where the vehicle is in front.

5. Monitor the lane change state of the ego vehicle that performs the lane change. The action should be blocked.

Scenario 4: Obstacles to the back and in front, both inside safety distances

Description: This scenario is the situation where the LCAI is active, and there are two obstacles inside the safety distance: one to the back and one to the front. It is shown in Figure 15.9. This aims to test whether the LCAI performs correctly when faced with multiple obstacles.

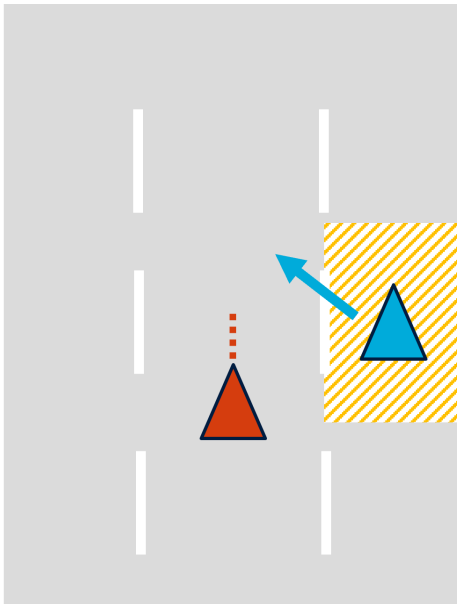


Figure 15.9: Vehicle next to ego within safety distance

How to replicate this scenario:

1. Initialize the simulation environment on a straight road with three vehicles side by side.
2. Move two vehicles onto the target lane, one slightly to the front of the ego vehicle that will perform the lane change and one slightly to the back.
3. Accelerate all vehicles to the same high speed in order to create a high safety distance.
4. Initiate the lane change to the lane where the two vehicles are driving.
5. Monitor the lane change state of the ego vehicle that performs the lane change. The action should be blocked.

Scenario 5: Obstacle to the back but with high relative speed

Description: This scenario is the situation where the LCAI is active and there is an obstacle outside the safety distance, but with a high relative speed, i. e. , it is approaching fast. This aims to test whether the LCAI blocks the lane change if there is a fast approaching obstacle to the back on the target lane. The scenario is shown in Figure 15.10.

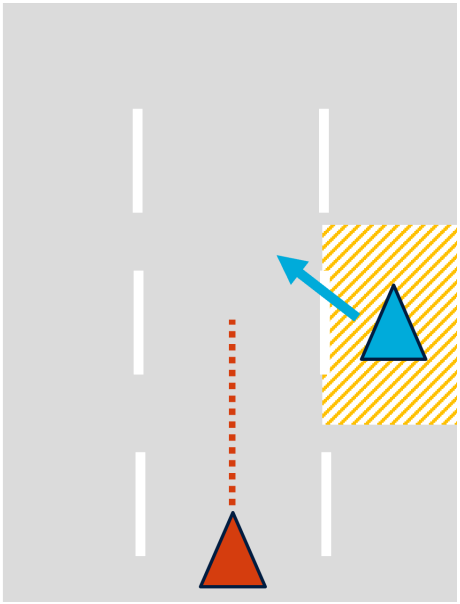


Figure 15.10: Fast approaching vehicle not yet within safety distance

How to replicate this scenario:

1. Initialize the simulation environment on a straight road with two vehicles side by side.
2. Move the vehicle on the target lane a great deal to the back of the ego vehicle that will initiate the lane change. Make sure that the relevant range for obstacle detection is not smaller than the distance created. Otherwise, the obstacle will not be noticed at all.
3. Accelerate the back vehicle to a significantly higher speed than the ego vehicle.
4. Without waiting for the behind vehicle to catch up, initiate the lane change.
5. Monitor the lane change state of the ego vehicle that performs the lane change. The action should be blocked, since the rear vehicle is fast approaching.

Scenario 6: Obstacle outside of safety distance

Description: This scenario is the situation where the LCAI is active and there is an obstacle outside the safety distance. This aims to test whether the LCAI recognizes correct safety distances and does not block the lane change. It is shown in Figure 15.11.
How to replicate this scenario:

1. Initialize the simulation environment on a straight road with two vehicles side by side.

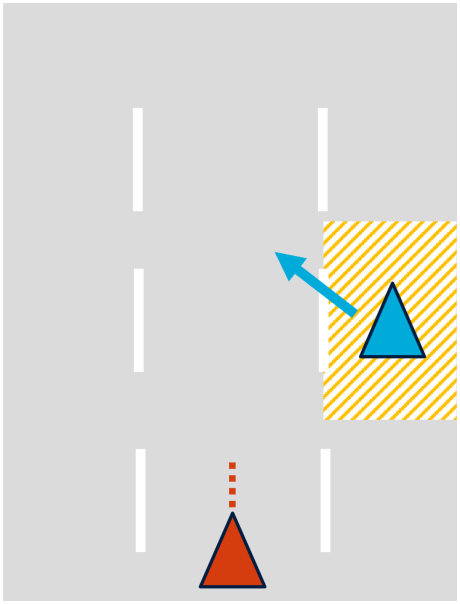


Figure 15.11: Slower moving vehicle outside the safety distance

2. Move the vehicle on the target lane a great deal to the back of the ego vehicle that will initiate the lane change.
3. Accelerate both vehicles to a high speed.
4. Initiate the lane change.
5. Monitor the lane change state of the vehicle that performs the lane change. The action should not be blocked, since the rear vehicle is outside the safety distance.

15.3 Obstacle Overtaking Safety and Road Rules Adherence

In this section, the adherence to overtaking safety and road rules is described. This is done similarly to the LCAI described in Section 15.2.

15.3.1 Introduction

The OTA also employs an Overtaking Assistant Interpreter (OTAI), just like the LCA. The interpreter architecture is described in detail in Subsection 14.1.2.

As stated in Section 15.2, the ego vehicle should not collide with obstacles during lane changes. This also applies to the lane changes performed using the OTA. However, the OTAI performs checks for more conditions than the LCAI, since the OTA operates in more complex situations than the LCA.

In addition to check for obstacles that might block the lane changing processes, the OTAI also checks for the following conditions:

I. Is overtaking prohibited by currently applicable road sign rules?

II. Is the speed difference to the obstacle big enough to justify an overtaking maneuver?

III. Would the ego vehicle be faster than the currently applicable speed limit during overtaking?

Also, checking the target lane for obstacles is not only relevant when performing the first OTA lane change, but also when reeving back onto the origin lane. In opposition, the LCAI only had to check one maneuver at all times.

In difference to the LCA, the OTA should not do nothing after being blocked, but remain behind the obstacle at a safe distance until the obstacle is overtakeable. The OTAI then frees the OTA, thanks to the supervisor architecture described in Subsection 14.1.2.

15.3.2 Implementation

For addressing lane changes, the OTAI simply uses the LCAI. The OTAI calls the inspection method of the LCAI, and uses its result for further interpretation of the situation. When the LCAI indicates that a lane change is not possible, the OTAI will not perform further checks, but informs the supervisor that the overtaking maneuver is not possible. The supervisor then blocks the OTA. See Section 15.2 for more details on the LCAI, and Subsection 14.1.2 for more details on the supervisor and interpreter architecture.

For checking whether overtaking is allowed or not, the OTAI uses the obstacle detection described in Chapter 10. The obstacle detection module recognizes road signs based on their ArUco ID, and adds certain rules into TurtleCars model if necessary. This concludes, that the OTAI can simply rely on the data in the model for checking road rules. If the OTAI finds a road sign rule which prohibits overtaking, it communicates this to the OTA, which is then blocked.

For checking if the relative speed difference of the ego vehicle to the obstacle is big enough to justify an overtaking maneuver, the information gathered by the obstacle detection module is used. All obstacles in the model have their relative speed and their absolute speed calculated and set. Therefore, the OTAI can simply access the relative speed and define a threshold, above which an OTA maneuver is possible.

A new requirement is, that the ego vehicle should not exceed the currently applied speed limit during the overtaking process. Otherwise, an overtaking maneuver could result in a very slow overtake, which is to be avoided. The OTA exposes the speed with which it would perform the overtake with. This speed is compared with the currently applicable speed limit. If the overtaking speed would exceed the speed limit, it is not performed until after the speed limit is lifted.

If any of the above conditions are violated, the OTAI indicates to the OTA that it can't be used, and the supervisor disables the OTA as long as the OTAI still blocks the OTA.

15.3.3 Scenarios

In this subsection, the scenarios that could occur with using the OTA are described, and whether these would lead to the OTAI blocking the OTA or not.

Figure 15.12 contains a scenario where the ego vehicle is blocked during usage of the OTA, because there is an obstacle on the lane which is used during overtaking. In Figure 15.13 however, there exists no such obstacle and the ego bot is therefore free to overtake using the OTA.

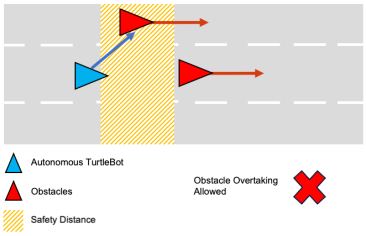


Figure 15.12: OTA blocked because of obstacles in overtaking lane

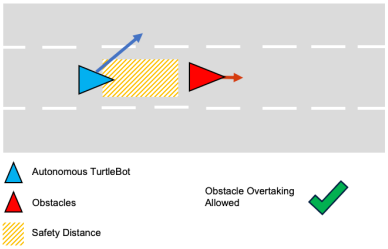


Figure 15.13: OTA free because of no obstacles in overtaking lane

In Figure 15.14, the ego vehicle is blocked because overtaking is prohibited in the ego vehicles current section. This road rule is lifted in Figure 15.15, so that the ego vehicle is free to overtake the obstacle with the OTA.

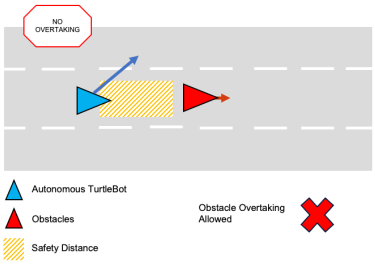


Figure 15.14: OTA blocked because of prohibited overtaking in section

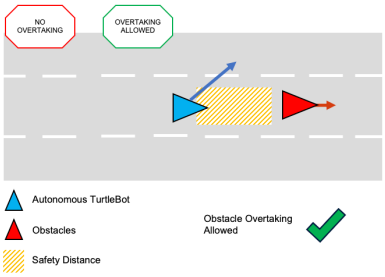


Figure 15.15: OTA free because of overtaking prohibition lifted again

The next scenario depicts a situation where the OTA is blocked because of having the same speed with the obstacle to overtake. In Figure 15.16, the speed difference for overtaking is not big enough in order to justify an overtaking maneuver. However, this is not the case in Figure 15.17: the speed difference is big enough and the ego vehicle is free to overtake.

Last, in the scenario depicted in Figure 15.18, the ego vehicles overtaking maneuver is blocked because the vehicle would reach a speed during the maneuver which is higher than the allowed maximum speed at this section. However, in Figure 15.19, the ego vehicle stays well under the maximum speed limit of $130\,\mathrm{km/h}$, so that the overtaking can be performed. In these scenarios, a delta of $10\,\mathrm{km/h}$ between current speed and overtaking speed has been assumed.

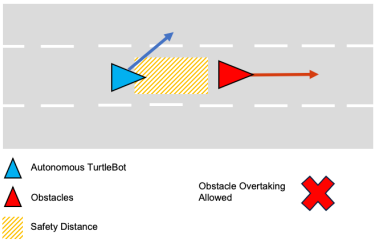


Figure 15.16: OTA blocked because of same speed with obstacle

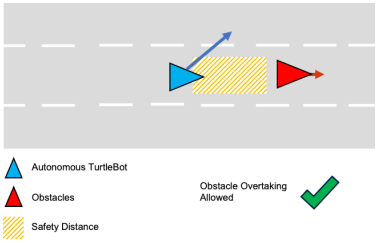


Figure 15.17: OTA free because of significant higher speed than obstacle

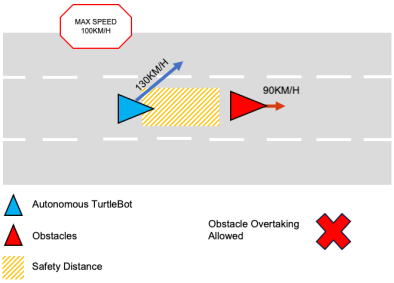


Figure 15.18: OTA blocked because of max speed limit applied during overtaking

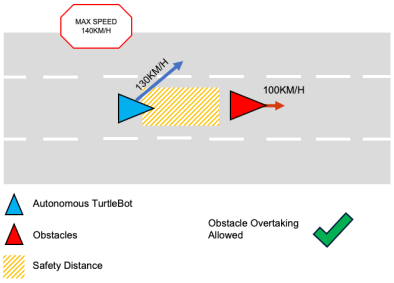


Figure 15.19: OTA free because of max speed not reached during overtaking

Chapter 16

Driving Functions

In this section, the driving functions implemented by the project group are documented. These are used within the architecture described in Chapter 14. For each driving function, the definition of the requirements, the controllers used to implement the driving function, and the validation are described. Some of these driving functions can't be active at the same time, as the controllers may provide conflicting control inputs. These conflicts are covered in Section 16.8

16.1 Manual Driving

The manual driving controller controls the vehicle conforming solely to the driver's inputs. Since the focus of the project group is the development of autonomous driving function and not a realistic mapping of controls of a real vehicle, the user input is simplified. It consists of two inputs: A target velocity and a steering wheel angle. The manual driving controller gives these inputs directly to the `Action` interface. This means that the steering angle is set to be exactly the steering wheel angle given by the driver and the target velocity of the vehicle is set to be exactly the target velocity given by the driver. These values are then used by the `Transposer` to drive the vehicle accordingly. It is necessary to mention that the `Transposer` acts as a very aggressive cruise control with maximum acceleration and deceleration. In order to enable more realistic and smoother driving, the input of the driver or the vehicle model needs to be changed.

16.2 Lane Keeping Assistant

The LKA driving function should assure that the vehicle keeps in its lane. In Figure 16.1 the ego vehicle is located on the middle lane and driving. In a scenario with the LKA activated the vehicle should keep the same distance to the lane markings on each side - so it should drive centered in the lane it starts in.

The requirements of the LKA are based on the ISO standard 11270 [31], but do not yet cover all aspects. These requirements are subject to rework.

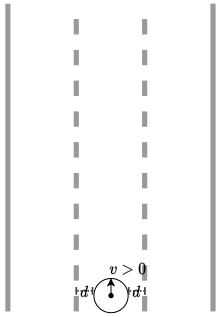


Figure 16.1: LKA scenario

16.2.1 General Requirements

- The LKA must be able to be switched on or off
  - The LKA can be toggled by user input
  - The LKA can be switched on by startup flag
- The robot must identify the lane its on and its center
- The LKA must be disabled when the lane change disabling condition according to the ALKS regulation is fulfilled
- The LKA enabled vehicle should never cross lane borders
- The robot should follow the lane's center
- The controller should be based on model prediction

16.2.2 Functional Requirements

Requirement LKA.1

GIVEN

- The vehicle is running

WHEN

- The input to enable the LKA is given

THEN

- The LKA is enabled

Requirement LKA.2

GIVEN

- The robot starts inside of lane boundaries

- The initial velocity is 0, the initial acceleration is 0, the initial steering angle is 0
- The robot heading fulfills the following criteria:
  - If the robot is left of the center of the lane, it faces in the direction it will drive, oriented within 0 and 40.2° toward the right lane boundary.
  - If the robot is right of the center of the lane, it faces in the direction it will drive, oriented within 0 and 40.2° toward the left lane boundary.

WHEN

- A target velocity greater than 0 is given by a human driver
- A steering wheel angle smaller than a threshold ω is given by the driver
- The LKA is enabled by the driver

THEN

- The LKA engages
- The robot identifies the lane it is on
- The robot accelerates to the speed defined by the human driver and maintains this speed
- The robot follows the center of the lane
- The robot never crosses lane borders
- The steering angle is always within the vehicle's specifications

Requirement LKA.3

GIVEN

- The robot is driving with arbitrary speed, arbitrary acceleration
- The LKA is active
- The steering angle is arbitrary within the vehicle's specification
- Initially there is no steering input from the user

WHEN

- A steering wheel angle greater than a threshold ω is given by the driver

THEN

- The LKA disengages
- The steering angle of the vehicle is the same as the steering wheel angle given by the driver

Requirement LKA.4

GIVEN

- The robot is driving with arbitrary speed, arbitrary acceleration
- The LKA is enabled
- The steering wheel angle given by the user is greater than a threshold ω
- The LKA is disengaged

WHEN

- A steering wheel angle smaller or equal to a threshold ω is given by the driver

THEN

- The LKA engages

Requirement LKA.5

GIVEN

- The LKA is enabled

WHEN

- The user input to disable the LKA is given

THEN

- The LKA is disabled

16.2.3 Non-Functional Requirements

LKA.A

The controller for the LKA is based on model prediction.

16.2.4 Additional Information

Overruling the LKA How an LKA can be overruled by the user differs depending on car manufacturer, model and available sensors and actuators in the car. In the manual for the EV6, KIA Motors describe that turning the steering wheel over a certain degree deactivates the LKA [40]. The deactivation criteria of the LKA systems developed by Bosch depend on the availability of power steering: When available, the LKA actively turns the steering wheel and can therefore be overruled by the driver using enough force. [72]). Since the TurtleCar system does not contain force feedback inputs and adding support for these is out of scope for this project group, this is not possible to implement. In order to demonstrate temporary overruling for a lane change, the steering wheel angle threshold ω is defined. It was determined experimentally that setting ω to 1.7° yields suitable results.

Determining the suitable initial conditions The maximum possible orientation is based on the most narrow curve that a car with the wheel base length and the maximum steering angle of a VW Golf. When in the center of the lane, the greatest angle it can recover from is 40.2°. This can be computed as follows:

- a: maximum steering angle (40°for VW Golf)
- w: wheelbase length (2.6365 m for VW Golf)
- r: radius of turning circle
- $r = \frac{w}{\tan(a)}$

Since the outer set of wheels on the car is of interest when determining the size of the circle driven by the vehicle, half of the golf's width is added to the to the radius:

$$r_{golf} = r + 0.9$$

The maximum recovery angle based on that radius was determined graphically as shown in Figure 16.2. When placing a circle with radius of r_{golf} so that it touches the lane boundary, it represents the path that a vehicle would take in the most extreme, still manageable case. The tangent of the circle at the intersection of the center of the lane shows the orientation of the car in this extreme case when placed at the lane center. When at the left of the lane center, the car is therefore able to recover from orientations of 53.6° to the right or lower. The same goes for the situation that the car is right of the center and faces left. With a safety margin of 25%, this results in a maximum allowed orientation of 40.2°.

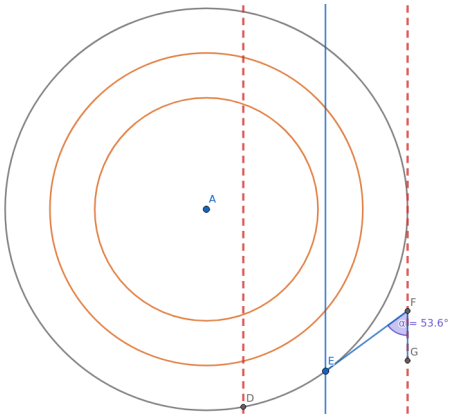


Figure 16.2: Graphical representation of the maximum heading pointing out of the lane that a car can recover from

16.2.5 Implementation

Two approaches were used to implement the LKA. The first approach is based on a PID controller, which is easier to implement but becomes unstable in some conditions. It was subsequently replaced by a controller based on MPC. Both approaches are presented in the following.

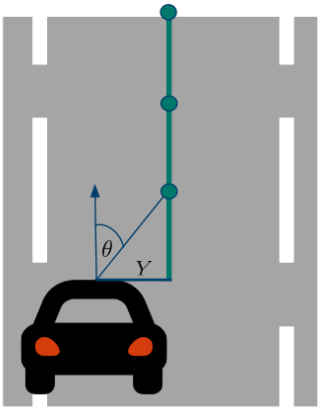


Figure 16.3: A graphical representation of the state variables used for the controller of the LKA. Y and θ are always relative to the next point provided by the path planner.

PID-based Lane Keeping Assistant

For the implementation, a MPC based on the bicycle model is used. Since the bicycle model is a nonlinear differential equation, it is linearized in order to obtain a linear controller.

Controller Model Since the LKA does not control the acceleration but only the steering angle, the velocity of the vehicle can be regarded as a parameter of the system. Therefore the bicycle model defined in Section 5.1 can be reduced to the simplified version

$$\dot{x_1} = \dot{Y} = v \cdot \sin(x_2)$$
$$\dot{x_2} = \dot{\theta} = \frac{v}{l} \cdot \tan(u_2)$$

where X is the position in the linear direction of the car, Y the lateral position, and θ the heading. These are all relative to the next point that the path planner provides. A depiction of the meaning of Y and θ can be seen in Figure 16.3.

This allows for a simpler linearization. The operating point to linearize around is $x = 0$ and $u = 0$. This represents the state where the vehicle is exactly on the line that has to be followed, and assumes that the controller only needs to make small corrections.

With that the system is linearized:

$$\begin{aligned}
\dot{x}(t) = f(x,u) = Ax + Bu &\approx f(0,0) + \frac{\partial f}{\partial x}\Big|_{\substack{u=0\\x=0}} \cdot x + \frac{\partial f}{\partial u}\Big|_{\substack{u=0\\x=0}} \cdot u \\
&= \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2 s} \end{bmatrix}\Bigg|_{\substack{u=0\\x=0}} \cdot x + \begin{bmatrix} \frac{\partial f_1}{\partial u_1} \\ \frac{\partial f_2}{\partial u_1} \end{bmatrix}\Bigg|_{\substack{u=0\\x=0}} \cdot u \\
&= \begin{bmatrix} 0 & v\cos(x_2) \\ 0 & 0 \end{bmatrix}\Bigg|_{x=0} \cdot x + \begin{bmatrix} 0 \\ \frac{v}{l \cdot \cos^2(u)} \end{bmatrix}\Bigg|_{u=0} \cdot u \\
&= \begin{bmatrix} 0 & v \\ 0 & 0 \end{bmatrix} \cdot x + \begin{bmatrix} 0 \\ \frac{v}{l} \end{bmatrix} \cdot u
\end{aligned}$$

The state feedback control law $u = -[k_1\ k_2] \cdot x$ is used to design the controller.

This results in the closed-loop function:

$$f_{cl} = \begin{bmatrix} 0 & v \\ 0 & 0 \end{bmatrix} \cdot x + \begin{bmatrix} 0 \\ \frac{v}{l} \end{bmatrix} \cdot (-[k_1\ k_2] \cdot x) = \begin{bmatrix} 0 & v \\ 0 & -\frac{v}{l} * k_2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

From the closed loop function it can be seen that k_1 can remain undetermined, as the lateral position has no direct influence on the control input. Now the operating domain for the velocity the LKA should be stable in has to be chosen. For this $v \in (0, 34]$ $^{\mathrm{m}}/_{\mathrm{s}}$ was chosen, which is $(0, 120]$ $^{\mathrm{km}}/_{\mathrm{h}}$.

Using Matlab the characteristic polynomial for which all eigenvalues have real parts strictly less than 2 was determined. With the coefficients, it is possible to solve for values of k_2 which hold the closed loop system in a stable domain. For this, the parameter space was sampled with a step size of 0.5. Since the system is uncontrollable for $v = 0$, sampling started at 0.5. This resulted in values for $k_2 \in [0.05, 1.4]$.

This enabled building a stable controller for the LKA. From the domain of stable values for k_2, choosing $k_2 = \frac{l}{v}$ has been determined experimentally to yield the best results for any speed v.

Model Predictive Control-Based Lane Keeping Assistant

The controller based on MPC is designed to follow a trajectory which leads along the center of the current lane. The trajectory is created in the same way as described in Section 16.2.5. It is updated every second to adhere changes in the lane detection.

The MPC-based controller to follow this trajectory is initialized with a step size of 1 second. The following weights for the error in each state dimension were chosen:

$$\begin{aligned}
Q_v &= 1.0 \\
Q_\psi &= 1.0 \\
Q_X &= 0.0 \\
Q_Y &= 1.0
\end{aligned}$$

This forces the MPC to adhere to the heading and lateral position Y and enables it to neglect the longitudinal position X which is less of interest as

staying in the lane only concerns the lateral movement. THe input error weights are evenly distributed:

$$R_a = 1.0$$
$$R_\omega = 1.0$$

From the result of the MPC, only the steering angle is forwarded to the vehicle input. Since the MPC is designed to choose both optimal acceleration and steering, it is necessary to constrain the acceleration to the value that the driver will probably choose, as it would lead to wrong results if the calculations within the MPC are based on different acceleration values. It is assumed that the driver will not perform any large speed changes and therefore the acceleration given by them is 0.0. Therefore, the MPC is constrained to have a minimum and maximum acceleration of 0.0 in every state.

16.2.6 Tests

Before the requirements are tested, a suitable environment is created. A scenario and the required number of bots are loaded for this purpose. Furthermore, areas, test phases and a timer are defined.

Turning the LKA on and off is one of the general requirements. If the LKA is activated, the robot should identify the line its centered on and follow the lane's center. Also the LKA enabled vehicle should never cross lane borders. In the designed test (auto_deactivation) the LKA is enabled. After five seconds the LKA is deactivated and the steering angle is set to a greater value than the LKA_DEACTIVATION_THRESHOLD. In this case the roboter cannot keep its lane, therefore the LKA is deactivated and the test is succesful.

Requirement LKA.1
The condition for a successful test (lka_drive_centered) is staying inside the designated center line for 10 seconds. The test fails, when the test condition isn't meet after 30 seconds.

Requirement LKA.2
Straight Pull: Success is determined if the vehicle is able to pull straight in time, demonstrating the LKA's ability to maintain lane centering after reactivation.

The test (lka_curved_road) is considered successful if the vehicle maintains its lane for more than 30 seconds. This duration tests LKA's sustained lane-keeping ability, addressing aspects of LKA.2, specifically maintaining lane position over time. Failure is recorded if the vehicle veers into either the left or right lane. These requirements ensure that the LKA functionality of a vehicle can effectively maintain the vehicle's position within its lane under various conditions.

Requirement LKA.3

The Requirement LKA.3 involves deactivating the LKA when the steering angle exceeds a certain threshold. Success Condition for the Test: The test is deemed successful if LKA is able to maintain lane integrity for a specified duration after reactivation. This indicates that the system can reliably re-engage

and continue to provide lane-keeping assistance even after it has been automatically deactivated under certain conditions. Failure Condition for the Test: The test is considered a failure if LKA fails to maintain lane integrity following its reactivation. This suggests that the system is not consistently reliable in re-establishing its lane-keeping capabilities after being deactivated due to excessive steering angle or other factors. This criterion is crucial for evaluating the robustness and reliability of the LKA system, especially in scenarios where temporary deactivation is necessary but should not compromise the vehicle's ability to return to and maintain its lane positioning.

Requirement LKA.4
LKA.4 concerns reactivating the LKA once the vehicle is correctly aligned within its lane (test named lka_reactivation). This criterion assesses the system's ability to re-engage and effectively support lane keeping after manual alignment by the driver or after any condition that required deactivation. A test is considered successful if, after reactivation, LKA maintains lane integrity for a predetermined amount of time. This outcome demonstrates the system's capacity to seamlessly reintegrate and uphold its lane-keeping functions after being temporarily disengaged. The test is deemed a failure if LKA is unable to maintain lane integrity following its reactivation. This indicates a deficiency in the system's capability to reliably resume its lane-keeping support after a period of deactivation, which is critical for ensuring continuous safety and assistance to the driver. This specification is vital for evaluating the effectiveness and reliability of the LKA system in scenarios where it has been deactivated and subsequently reactivated, ensuring that the system can still provide consistent support for maintaining the vehicle within its lane. In the test (pull_straight) the reactivation of LKA after manual steering is being tested.

Requirement LKA.5
To manually test the deactivation of the LKA, first it is required to deactivate LKA and then adjust the steering to be less than the LKA Deviation Threshold. This procedure aims to simulate conditions under which LKA might be intentionally turned off or adjusted due to driver input or specific driving scenarios (test: lka_manual_activation) If the robot is partially within the left lane, it is an indication that LKA has been successfully disabled (LKA.5). This outcome confirms the system's responsiveness to manual deactivation commands and its ability to correctly interpret and react to steering adjustments that fall below the set deviation threshold.

This test is crucial for assessing the LKA system's flexibility and its capacity to be manually overridden or adjusted by the driver, ensuring that control can be seamlessly transferred without compromising safety or driving efficiency.

LKA System Validation Summary
The testing and validation of the LKA system confirmed its successful alignment with all predefined requirements(Requirements ACC.1 through ACC.5), effectively ensuring the system's ability to keep the vehicle centered within its lane, respond to manual driver inputs for activation and deactivation, and adaptively maintain lane integrity under various driving conditions.

16.3 Adaptive Cruise Control

The ACC function should assure that a vehicle keeps a safety margin to a vehicle in front of it. In Figure 16.4 the ego vehicle is located on the middle lane and driving. In front of the ego vehicle is another vehicle, driving at least as fast as the ego vehicle. The distance between both vehicles is at least the safety margin distance at all times.

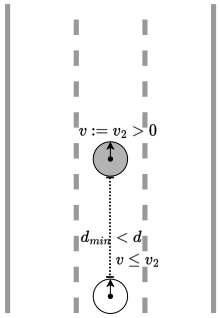


Figure 16.4: ACC scenario

16.3.1 General Requirements

- The vehicle must be able to activate/deactivate ACC depending on the drivers wishes
  - The ACC must be deactivated if the driver takes manual control
- The vehicle must keep at least a minimum safe distance including a suitable error margin
  - The vehicle should keep the same speed as the front vehicle
  - The vehicle must be able to reduce its speed to keep the minimum safe distance
  - The vehicle should be able to increase its speed to keep the distance to front vehicle
- The driver can issue a command to drive at a certain speed overriding the ACC
- The ACC must be disabled if the driver issues a brake command
- The ACC controller should be build using model prediction

Definition: Minimum Safe Distance To be defined according to relevant regulations: Minimum safe distance which a following vehicle needs to maintain in order to be able to decelerate if the leading vehicle brakes (with bounds for deceleration)

In the German traffic regulations, a rule of thumb to determine the distance between two vehicles considered safe is described: The vehicle needs to keep a distance of at least half of the current speed (kilometers per hour) in meters [14, §2 Abs. 3a S. 2a].

16.3.2 Functional Requirements

Requirement ACC.1

GIVEN

- The ego vehicle is driving behind another vehicle. Both vehicles have arbitrary speed and the ego vehicle maintains at least minimum safe distance to the leading vehicle.

WHEN

- The driver triggers the switch for the ACC

THEN

- The ACC is enabled

Requirement ACC.2

GIVEN

- The vehicle is driving behind another vehicle with arbitrary speed v_i with at least minimum safe distance including error margin

WHEN

- The ACC is enabled

THEN

- The ego vehicle drives at most with velocity v_i
- The ego vehicle accelerates or brakes within the velocity bounds so that it maintains at least a minimum safe distance including error margin to the leading vehicle

Requirement ACC.3

GIVEN

- The ACC is enabled
- The vehicle is driving behind another vehicle with arbitrary speed v_i and has at least minimum safe distance including error margin

WHEN

- The other vehicle brakes to velocity v_b

THEN

- The ego vehicle drives at most with velocity v_b
- The ego vehicle decelerates to velocity v_b and maintains at least a minimum safe distance without error margin at all times

Requirement ACC.4

GIVEN

- The ACC is enabled
- The leading vehicle is driving with velocity v_i
- The vehicle is driving behind another vehicle with arbitrary speed v_i and has a a distance to the leading vehicle that is smaller than the minimum safe distance including error margin
- The driver does not give a command to accelerate to a speed greater than v_i

WHEN

- No Action

THEN

- The ego vehicle drives at most with velocity v_i
- The ego vehicle decelerates to until it maintains at least a minimum safe distance including error margin

Requirement ACC.5

GIVEN

- The ACC is enabled
- The vehicle is driving behind another vehicle with arbitrary speed v_i and has at least minimum safe distance including error margin

WHEN

- The other vehicle accelerates to velocity v_a

THEN

- The ego vehicle drives at most with the minimum v_m of velocities v_i and v_a
- The ego vehicle accelerates to velocity v_m and maintains at least a minimum safe distance with error margin at all times

Requirement ACC.6

GIVEN

- The ACC is enabled
- The vehicle is driving behind another vehicle with arbitrary speed v_i and has at least minimum safe distance including error margin

WHEN

- The driver continuously issues a command to drive with velocity v_t

THEN

- The ego vehicle accelerates to velocity v_t without regard for the minimal safe distance

Requirement ACC.7

GIVEN

- The ACC is enabled

WHEN

- The relevant user input is received

THEN

- The ACC is disabled
- The vehicle drives according to the driver's commands only

Requirement ACC.8

GIVEN

- The ACC is enabled

WHEN

- The driver issues a braking command

THEN

- The ACC is disabled
- The vehicle drives according to the driver's commands only

16.3.3 Non-Functional Requirements

ACC.A

The controller for the LKA is based on model prediction.

16.3.4 Implementation

For the implementation, the approach described by Zhenhai et al. is used [98]. The function of the ACC is based on a switching strategy between two modes: In Cruise mode the vehicle simply keeps a given speed. In Follow mode the vehicle maintains a constant distance to the preceding vehicle and matches its speed if it is lower than the speed defined by the driver. The control law in each mode defines the acceleration a. For the description of the controllers, the following variables are used:

- v is the speed of the ego vehicle
- v_d is the target speed for the ACC
- v_p is the speed of the preceding vehicle
- $\Delta v = v_p - v$ is the relative speed of the two vehicles
- $\Delta d = d - d_{\min(\Delta v)}$ is the distance of the ego vehicle to the closest safe point behind the preceding vehicle based on their relative speed
- a_{follow} is the target acceleration calculated by the follow mode algorithm
- a_{cruise} is the target acceleration calculated by the cruise mode algorithm
- d_{offset} is a parameter to define the offset between the zones where Follow mode and Cruise mode are applied

The control laws in each mode are described in Table 16.2. The constants k_p, k_i, k_v and k_d are selected so that the vehicle is able to achieve the desired state quickly, but without creating too much jerk in its movements. The chosen values are shown in Table 16.1.

Table 16.1: Chosen values for ACC constants.

Constant	Value
k_p	0.5
k_i	0
k_v	0.25
k_d	1

Table 16.2: The control laws of the ACC.

	Control Law
Cruise Mode	$a_{cruise} = k_p(v_d - v) + k_i \int (v_d - v)dt$
Follow Mode	$a_{follow} = k_v \Delta v + k_d \Delta d$

Each time a control signal needs to be generated, the controller first checks which mode should currently be applied. It then chooses the corresponding control law. This switching strategy is taken from Zhenhai et al. and is based on dividing the parameter space into zones in which the different control

Table 16.3: The zone-based switching strategy from Zhenhai et al. [98]

Distance	Velocity	Acceleration	Resulting Control Mode
$\Delta d \leq 0$	$\Delta v \leq 0$	-	Follow Mode
$\Delta d > 0$	$\Delta v < 0$	$a_{follow} \leq a_{cruise}$	Follow Mode
$\Delta d > 0$	$\Delta v < 0$	$a_{follow} > a_{cruise}$	Cruise Mode
$\Delta d < d_{offset}$	$\Delta v > 0$	$a_{follow} \leq a_{cruise}$	Follow Mode
$\Delta d < d_{offset}$	$\Delta v > 0$	$a_{follow} > a_{cruise}$	Cruise Mode
$\Delta d \geq d_{offset}$	$\Delta v \geq 0$	-	Cruise Mode

laws apply [98]. This division is given by distance, velocity and acceleration. This method ensures that switching between modes is conducted smoothly. In favor of brevity, only the switching table is documented here without reasoning about the parameter zones. It is shown in Table 16.3. The table describes the conditions on distance, velocity and acceleration and defines the control mode that should be applied in each case. A „-“ means that the respective parameter does not factor in the decision in this case.

16.3.5 Tests

Requirement ACC.1 & Requirement ACC.2 & Requirement ACC.4
The test (acc_active) assesses the ACC system under the conditions specified in Requirement ACC.2 and ACC.1, focusing on its capacity to maintain safe following distances and adjust speed in response to the lead vehicle's behavior.

The test is considered successful if, upon ACC activation, the ego vehicle adjusts its speed to not exceed that of the leading vehicle (velocity v_i), and actively manages its speed (either by accelerating or braking) to maintain a minimum safe distance, inclusive of an error margin, from the vehicle ahead. This demonstrates the ACC system's effectiveness in adapting to varying speeds of the leading vehicle while ensuring a consistent safety buffer is maintained.

The test is deemed a failure if the ego vehicle fails to adhere to the speed of the leading vehicle (exceeds velocity v_i) or does not maintain the minimum safe distance, considering the error margin. Failures indicate deficiencies in the ACC system's responsiveness or its ability to calibrate speed and distance controls accurately under specified conditions.

Another test scenario (acc_self_decelerate) also examines the ACC's capability to maintain a safe following distance when activated, closely aligning with Requirement ACC.2 and ACC.1. Initially, the ACC is not active, simulating normal driving conditions. After a brief period, the ACC is activated, and the system's ability to manage the vehicle's speed to maintain or achieve a safe distance from the vehicle ahead is tested.

Success is achieved if the ego vehicle adjusts its speed to maintain at least the minimum safe distance from the leading vehicle, demonstrating the ACC's effectiveness in ensuring safety and compliance with the set distance parameters.

The test fails if the ego vehicle does not maintain the minimum safe distance within the test duration, indicating a potential issue in the ACC system's distance sensing or speed adjustment capabilities.

Both tests also share similarities with Requirement ACC.4. Requirement

ACC.4 specifically addresses how the ACC system responds when the vehicle is initially too close to the vehicle ahead, requiring deceleration to establish a safe following distance, without driver acceleration commands. In contrast, ACC.1 focuses on the activation of ACC, and ACC.2 outlines the system's operation to maintain or achieve a safe distance at all times.

Requirement ACC.3
The scenario (acc_other_decelerate) tests the ACC system's response to a leading vehicle's deceleration (Requirement ACC.3). The test initiates with both vehicles moving and the ACC activated on the ego vehicle. The scenario progresses to simulate a deceleration of the lead vehicle, examining the ego vehicle's capability to adjust its speed accordingly to maintain a safe following distance.

The test is deemed successful if the ego vehicle decelerates to match the lead vehicle's reduced speed without breaching the minimum safe distance. This outcome signifies the ACC system's effective speed adjustment and distance management capabilities in response to the changing speeds of the vehicle ahead, reflecting compliance with ACC.3.

A failure occurs if the ego vehicle either fails to decelerate adequately or maintains a distance to the lead vehicle that falls below the minimum safe distance threshold. Such outcomes indicate shortcomings in the ACC system's ability to dynamically adapt to deceleration events in the traffic flow.

Requirement ACC.5
Another test (acc_other_accelrate) evaluates the ACC system's capability to adjust the ego vehicle's speed in response to the lead vehicle's acceleration (Requirement ACC.5). After initiating the scenario with both vehicles in motion and the ACC activated, the lead vehicle's speed is increased. The focus is on the ego vehicle's ability to autonomously accelerate in order to maintain a consistent distance to the lead vehicle.

Success is achieved if the ego vehicle accelerates beyond a specified speed threshold, indicating effective tracking and response to the lead vehicle's increased speed. This demonstrates the ACC system's proficiency in adapting to dynamic traffic conditions and maintaining safe, consistent following distances.

Failure occurs if the ego vehicle does not adjust its speed accordingly within the test duration, suggesting a shortfall in the ACC system's responsiveness or its ability to maintain the specified safe distance under changing conditions.

Requirement ACC.6
This scenario (acc_self_accelerate) tests the ACC's responsiveness to driver commands for acceleration, as specified in Requirement ACC.6. It starts with the vehicle initially accelerating to a nominal speed, then simulates the driver's command for further acceleration beyond this initial setting.

The test is successful if the ego vehicle accelerates to the specified higher velocity, demonstrating the ACC's ability to prioritize driver commands over maintaining preset safe distances.

If the vehicle fails to reach the commanded speed within the allocated timeframe, the test fails, indicating the ACC's potential overemphasis on distance control at the expense of driver intent.

This test is crucial for assessing the ACC system's flexibility in integrating driver preferences.

Requirement ACC.7
Requirement ACC.7 specifies the deactivation of the ACC system upon receiving relevant user inputs, allowing the vehicle to revert to manual control per the driver's commands. This requirement emphasizes the importance of driver control and the system's adaptability to seamlessly switch between automated and manual driving modes, contrasting with the focus of ACC.1 and ACC.2 on system activation and maintaining safe distances, and ACC.4's emphasis on system response to close proximities without driver acceleration commands. This is effectively tested by simulating a situation where the driver decides to manually override the ACC's controls, leading to its deactivation.

Requirement ACC.8
The test (acc_deactivation) assesses the ACC's deactivation in response to a driver's braking command, as specified in Requirement ACC.8. It begins with the ACC activated and the vehicle in motion. The scenario then simulates a braking command by altering the target speed. Success is determined by the vehicle's compliance with this new speed setting, effectively demonstrating the ACC's deactivation and the vehicle's manual control. This test underscores the importance of ACC responsiveness to ensure driver control and safety.

ACC System Validation Summary
The implementation of the ACC system testing and validation, successfully meeting all outlined requirements (Requirements ACC.1, ACC.2, ACC.2, ACC.3, ACC.4, ACC.5, ACC.6, ACC.7, ACC.8), thereby ensuring its capability to maintain safety margins, adapt to dynamic traffic conditions, and respond to driver inputs for activation, deactivation, and manual control overrides.

16.4 Lane Change Assistant

The LCA driving function (LCA) enables the vehicle to automatically and safely change lanes without the need for a driver to perform manual steering. The driver is able to press a button which indicates into which lane they would like to change. The driving function then checks whether the lane change into the target lane can be safely performed, i. e. , whether there are obstacles or other vehicles in the way, and perform the lane change if possible. Steering commands of the driver that exceed a certain threshold deactivate the LCA, so the driver can break off the maneuver by turning the steering wheel. The LCA can also be deactivated by pressing a button. In contrast to the LKA, after manually overruling the LCA by turning the steering wheel far enough, it must be manually re-initialized by the driver so that the car does not perform any unforeseen steering maneuvers after the steering commands of the driver no longer exceed the threshold.

In Figure 16.5 an example is given where the ego vehicle is driving on the middle lane. After the LCA was given the command to drive to the left lane by the driver, the vehicle should at some point in time drive on the left lane — having performed a lane change.

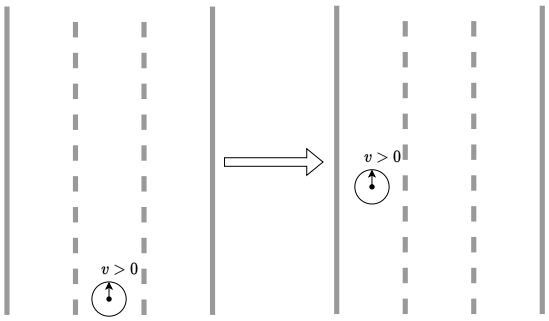


Figure 16.5: LCA scenario

16.4.1 General Requirements

- The LCA must be able to be switched on or off
  - The LCA can be toggled by user input
  - The LCA can be switched on by startup flag
- The ego vehicle must identify when a lane change is needed
- The LCA must ensure the ego vehicle only changes lanes when safe to do so, adhering to velocities and the StVO
- The ego vehicle must stay in its current lane if a lane change is not possible.
- The ego vehicle must execute a lane change as soon as it's safe
- The controller should be based on model prediction to ensure smooth and safe lane changes
- During a lane change, the ego vehicle must
  - Maintain a minimal safety distance
  - Use steering angles fitting to its current speed
- The driving function of the LCA must deactivate after a successful lane change or if it is deactivated manually

16.4.2 Functional Requirements

Requirement LCA.1
GIVEN

- The ego vehicle is operating at a constant velocity

WHEN

- The initial lane is confirmed to be empty and is expected to remain so until the lane change is executed

THEN

- The LCA remains engaged, monitoring for the possibility to change lanes

Requirement LCA.2

GIVEN

- The ego vehicle is operating in driving mode, with LCA engaged

WHEN

- The LCA identifies that a lane change can be executed
- There is no car in the lane to change to, or if there is, the velocities and the StVO permit the lane change

THEN

- The LCA executes the lane change while ensuring
-
  - The ego vehicle adheres to a minimal safety distance
  - The ego vehicle uses steering angles fitting to its current speed

Requirement LCA.3
GIVEN

- The LCA is engaged.
- A lane change has been initiated

WHEN

- The lane change is completed successfully

THEN

Requirement LCA.4

- The driving function of the LCA deactivates automatically

GIVEN

- The LCA is engaged

WHEN

- The LCA is deactivated manually by the operator

THEN

- The LCA ceases to initiate or continue a lane change, and the ego vehicle continues to drive in the current lane

16.4.3 Implementation

The LCA is based on the MPC described in Section 14.2. Its function is to create a trajectory for the lane change. The trajectory is constructed using a timestep of 0.1 s. The LCA can be in one of four states:

1. Inactive
2. Changing
3. Pulling Straight
4. Completed

How the LCA switches between these states is shown in Figure 16.6. Before activation, it is always in the Inactive state. When activating it with a direction (left or right), the LCA switches into the Changing state. Here, the lane change maneuver is planned and executed. At each step, it checks whether the lane change is complete, i. e. , whether the vehicle is in the target lane and has a heading that is pointing along the target lane within 0.05 rad.

If this is the case, the LCA changes into the Completed state, from which it directly changes into the Inactive state and deactivates itself. If the vehicle is in the target lane and does not have a heading that is within the error bounds around the heading of the target lane, but there are less than 10 steps left in the trajectory, the LCA changes into the Pulling Straight state and appends the trajectory by creating a trajectory that follows the lane center. It stays in this state until the heading is within the error bounds and then changes into the Completed state, from where it changes to the Inactive state and deactivates itself.

Within the Changing state, the actual planning of the lane change trajectory is conducted. The trajectory is re-planned once every second to ensure that current information about the environment is included. The trajectory consists of three sections.

The first section is a path through the current lane at the current offset of the car with respect to the lane. This section is 100 meters long. The second section is a straight path that starts at the end of the first section and goes at an angle towards the target lane. It ends at the point where it intersects the target lane's center. The angle is relative to the first lane and is velocity-dependent. Slower speeds lead to a steeper angle than faster speeds. The formula to calculate the relative angle is $\max(C, K - v)$ where C is the minimum angle, K is the maximum angle and v is the current velocity in meters per second. The third section starts at the end of the second section and goes along the center of the target lane for 500 meters.

The path consisting of these three sections is a list of points without additional information. These are transformed into a full state trajectory where the velocity in each state is set to be the current velocity and the acceleration is set to be 0 in all states. This is based on the assumption that the driver will not make any significant velocity changes during the lane change maneuver. The resulting trajectory is then transformed into an MPC problem and solved to obtain the next control signal. From these, only the steering angle is given to the vehicle, as controlling the acceleration is not the task of the LCA. When re-planning the trajectory, the sections that have already passed are discarded

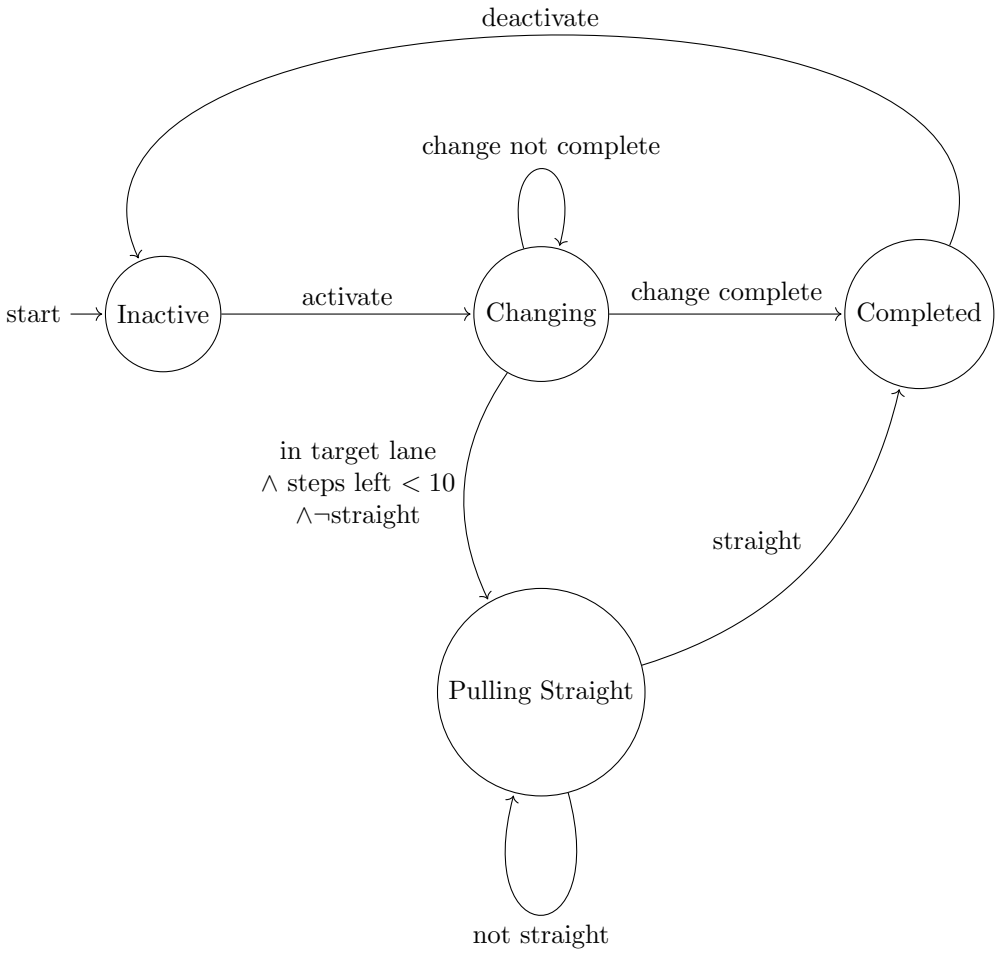


Figure 16.6: The states and state changes of the LCA.

and the planning starts with the current section and from the current point of the vehicle. The current section is planned only for the remaining number of steps in that section.

16.4.4 Tests

A scenario is loaded where the ego vehicle is initialized on an empty road. This environment is devoid of any other vehicles to simulate an unobstructed condition for a lane change. A timer is started to measure the execution time of the test.

Turning the LCA on and off is a general requirement for the ego vehicle. When the LCA is engaged, it is expected that the ego vehicle will identify when a lane change is necessary and execute it if safe to do so. The designed test (lca_basic) enables the LCA. The ego vehicle will change lanes to the left, given the initial lane remains empty.

Requirement LCA.1 The ego vehicle should initiate the lane change immediately after the velocity threshold is reached. The condition for a successful test (lca_basic) is completing the lane change within 30 seconds without any obstructions. The test fails if the ego vehicle cannot change lanes or if the change takes longer than 30 seconds to complete.

Requirement LCA.2 The test (lca_behind_blocked) is designed to ensure that the ego vehicle does not change lanes when there is an obstacle behind it within a certain safety distance. It demonstrates the LCA's compliance with safety regulations by not allowing a lane change when it would result in a dangerous situation. Success is determined if the ego vehicle remains in the original lane without attempting a change. Failure is indicated if the ego vehicle changes lanes and breaches the minimum safety distance, thus failing to uphold the set safety standards. The test (lca_front_blocked) is similar.

Requirement LCA.3 The LCA is tested for its ability to automatically disengage after a successful lane change.

The test (lca_basic) is considered successful if the ego vehicle executes the lane change within the set parameters and the LCA disengages automatically. This assesses the system's ability to identify the completion of the lane change maneuver and transition back to regular driving mode.

A failure in this test occurs if the ego vehivle does not complete the lane change or if the LCA does not automatically disengage after the lane change is completed. The LCA's performance in returning the vehicle to standard operation without manual intervention is critical for user convenience and system reliability.

Requirement LCA.4 This requirement examines the LCA's responsiveness to manual deactivation commands during a lane change process.

The test (lca_deactivation) is successful when the LCA ceases the lane change immediately upon manual deactivation by the operator and the ego vehicle continues in its current lane without attempting to complete the lane change.

Failure to comply with the manual deactivation command or continuing the lane change despite the command will result in a test failure. This test ensures that the LCA respects driver inputs and can be overridden, providing essential control back to the operator when needed.

Additional Test Scenarios

(lca_front_blocked): Assesses the LCA's decision-making when the path in the desired lane is obstructed. (lca_no_deactivation): Verifies that the LCA does not disengage prematurely when encountering minor steering disturbances. (lca_not_blocked): Confirms that the LCA proceeds with a lane change when the path is clear and safe, without being overly sensitive to distant vehicles. (lca_speed_blocked): Tests the LCA's ability to detect and respond to fast-approaching vehicles, preventing unsafe lane changes. (lca_behind_blocked): Ensures that the LCA does not initiate a lane change when there is a vehicle close behind, maintaining a safe following distance. Each test scenario aligns with the goal to ensure that the LCA functionality can safely manage lane changes, respecting the system's autonomous decision-making and also the driver's manual commands.

LCA System Validation Summary

The implementation of the LCA system underwent comprehensive testing and validation, successfully meeting all outlined functional requirements (Requirements ACC.1 through ACC.8), thereby ensuring its capability to assist drivers in safely changing lanes, adapting to traffic conditions, and responding accurately to driver commands for system activation and deactivation.

16.5 Obstacle Avoidance

The OTA driving function ensures a safe overtaking maneuver when a static obstacle is ahead. In Figure 16.7 the ego vehicle is located on the middle lane and driving. In front of it — in a safe distance — an obstacle is located. In a scenario with the OTA activated, this minimum distance is kept and a lane change is performed in order to overtake the obstacle and resume with the speed specified by the driver. The ego vehicle changes back to the middle lane at some point so that a safe distance from the obstacle is assured. Lane changes needed for an overtaking maneuver are considered safe as defined by the LCA requirements in Subsection 16.4.2.

In the context of this project group, „obstacle avoidance“ originally refered to the avoidance of **static** obstacles, whilst „obstacle overtaking“ originally only referred to the overtaking of **dynamic** obstacles. These two aspects were planned seperately in the beginning and received individual requirements. However, both functionalities were eventually implemented in one shared component, the OTA. Due to this, the term overtaking is now effectively used for the maneuver in which a static or a dynamic obstacle is passed.

16.5.1 Requirements

Requirement OTA.1

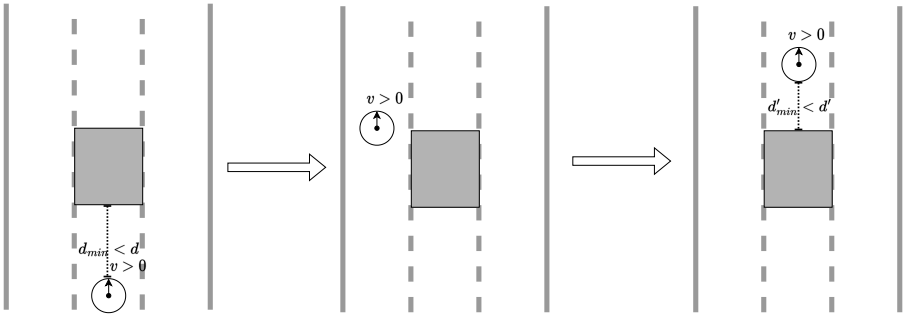


Figure 16.7: Obstacle Avoidance scenario

GIVEN

- The OTA is enabled
- There is an obstacle in front of the ego vehicle coming closer, i. e. the distance between the ego vehicle and some obstacle in the same lane approaches the minimum safe distance.

WHEN

- Occupancy of the left lane does not allow for a safe lane change.

THEN

- The ego vehicle brakes (eventually stopping) to maintain at least a minimum safe distance with error margin from the obstacle at all times.

Requirement OTA.2

GIVEN

- The OTA is enabled
- There is an obstacle in front of the ego vehicle coming closer, i. e. the distance between the ego vehicle and some obstacle in the same lane approaches the minimum safe distance.

WHEN

- Occupancy of the left lane allows for a safe lane change.

THEN

- The ego vehicle initiates an overtaking maneuver by changing lanes to the left.

Requirement OTA.3

GIVEN

- The OTA is enabled and an overtaking maneuver was started by a lane change to the left.
- There is at least one obstacle in the lane to the right of the ego vehicle.

WHEN

- Occupancy of the right lane does not allow for a safe lane change to reeve back into the original lane.

THEN

- The ego vehicle remains in its lane and keeps its velocity.

Requirement OTA.4

GIVEN

- The OTA is enabled and an overtaking maneuver was started by a lane change to the left.
- There is at least one obstacle in the lane to the right of the ego vehicle.

WHEN

- Occupancy of the right lane allows for a safe lane change to reeve back into the original lane.

THEN

- The ego vehicle concludes the overtaking maneuver by changing lanes to the right.

16.5.2 Implementation

Due to similarities in their requirements, it was decided to treat the avoidance of a static obstacle as a special case of a regular overtaking maneuver. This allowed for a single implementation that works in both scenarios. Refer to the next section for more details.

16.6 Overtaking

The OTA driving function also ensures a safe overtaking maneuver when a moving obstacle is ahead. In Figure 16.8 the ego vehicle is driving in the middle lane. In front of the ego vehicle is another vehicle driving. The distance between both vehicles is at least the safety margin distance. In a scenario with the OTA activated, this minimum distance is kept and a lane change performed. The ego vehicle changes back to the middle lane at some point so that a safe distance from the other vehicle is assured.

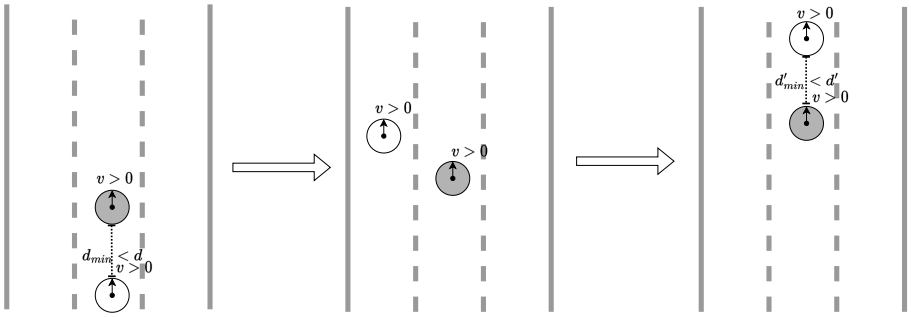


Figure 16.8: Overtaking scenario

16.6.1 Requirements

Requirement OTA.5

GIVEN

- The OTA is enabled
- The vehicle is driving behind another vehicle and has at least minimum safe distance including error margin

WHEN

- Overtaking is forbidden regarding the currently active road rules.

THEN

- The lane change to the left of the overtaking maneuver is blocked.

Requirement OTA.6

GIVEN

- The OTA is enabled
- The speed limit is v_m
- The vehicle is driving behind another vehicle with velocity v_b and has at least minimum safe distance including error margin

WHEN

- $v_m - v_b < 20^{\mathrm{km}/\mathrm{h}}$

THEN

- The lane change to the left of the overtaking maneuver is blocked.

16.6.2 Implementation

The OTA was designed to reuse the existent LKA, LCA and ACC. This decision was made since these three components already provide most of the needed functionality and are thoroughly tested, thus reducing complexity. An alternative approach that was considered involved the implementation of the OTA with its own MPC. This approach offers the possibility to reduce the whole overtaking maneuver to a single trajectory to be solved, but prevents the internal usage of the other driving functions, which is why it was ultimately discarded.

When the OTA is activated, the ACC is also always active until the maneuver is finished, since the ego vehicle should maintain both its safety distance to obstacles in front of it as well as a high speed at all times. Beyond that, the overtaking process can be divided into the following five states, with the transitions between these states being depicted in Figure 16.9.

1. **Inactive:** The OTA is enabled but the conditions to overtake are not met yet.

2. **Change Left:** The ego vehicle starts the overtaking maneuver by changing to the left lane.

3. **Overtake:** The ego vehicle maintains high speed to quickly pass the obstacle(s) on the right lane.

4. **Change Right:** The ego vehicle reeves back into its original lane respecting safety distances.

5. **Completed:** The overtaking maneuver is completed.

In the initial state, the LKA is used in addition to the ACC, such that the ego vehicle either maintains the current lane and specified speed if possible, or stays closely behind an obstacle in front of it until the actual overtake begins. If the OTA then retreives the information that an overtake is possible from the OTAI (see Section 15.3), it switches to the Change Left state and uses the LCA. From there, a signal from the LCAI (see Section 15.2) indicates the finished change, which results in a switch to the Overtaking state. The ACC and LKA then simply maintain the current lane and speed (if possible) for as long as the right lane is occupied. After passing the obstacles (including safety distance), the OTAI signals that the ego vehicle can reeve back and the OTA switches to the Change Right State, using the LCA again. As soon as the change is completed the overtaking maneuver is considered done and the OTA awaits its deactivation.

16.6.3 Tests

Requirement OTA.1 The dedicated test (ota_obstacles) validates the conditions from Requirement OTA.1. For that, additional to a static obstacle in front of the ego vehicle that should be overtaken, another static obstacle is placed on the left lane besides the ego vehicle. The test is considered to be successful if the ego vehicle starts an overtake maneuver only after the obstacle on the left is passed.

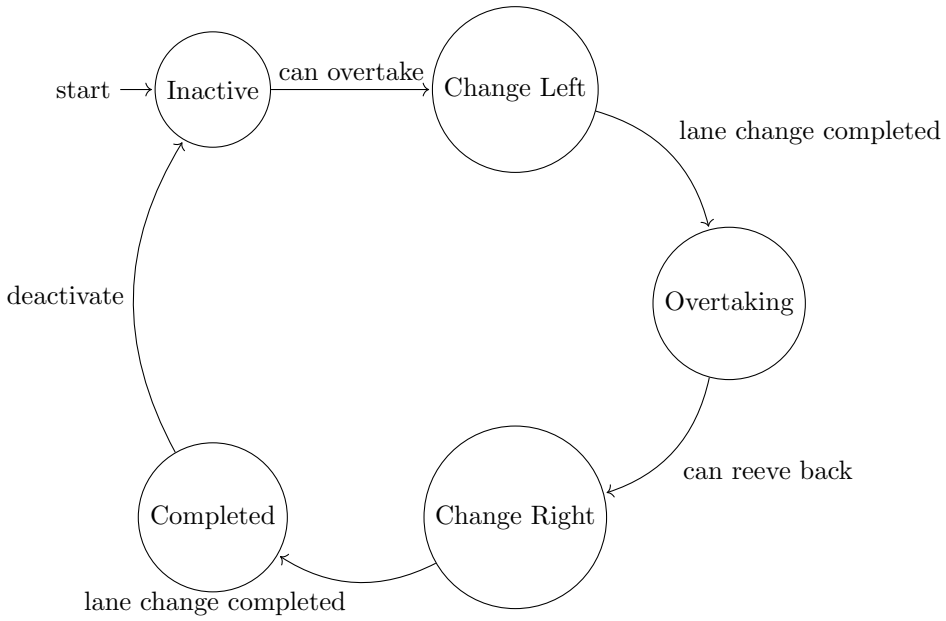


Figure 16.9: The states and state changes of the OTA

Requirement OTA.2 & Requirement OTA.3 & Requirement OTA.4 These requirements are validated by multiple tests (ota_dynamic, ota_obstacles and ota_static). They aim to provide a variety of situations that allow for a complete overtake, using both static and dynamic obstacles in front or besides the ego vehicle. All of these tests are considered sucessful if the ego vehicle first does a change to the left lane and then does another change to the right lane as soon as the obstacle on the right lane is passed. An additional test (ota_blocked_other_vehicle) checks whether the OTA is also correctly blocked when another vehicle on the left lane approaches too fast to enable for a safe first lane change.

Requirement OTA.5 Another test (ota_blocked_sign) validates the adherence to the „no overtaking“ road sign. It uses a world with two road signs for this purpose. The initial sign creates a „no overtaking“ zone, such that the ego vehicle is forced to drive behind the slower vehicle in front, without attempting to overtake it. The second sign annulates the previous „no overtaking“ rule. The test is therefore considered sucessful if the ego vehicle starts an overtake maneuver only after passing the second sign.

Requirement OTA.6 This requirement is tested analogously to the previous one. Its dedicated test (ota_blocked_speedlimit) also uses two signs, where the first one conducts a speed limit of $80^{\mathrm{km}/\mathrm{h}}$. Since the vehicle in front of the ego vehicle does not drive with over $20^{\mathrm{km}/\mathrm{h}}$ less than the ego vehicle, the overtake should be blocked until the second sign annulates the speed limit. The test is therefore considered sucessful if the ego vehicle starts an overtake maneuver only after passing the second sign.

OTA System Validation Summary
The testing and validation of the OTA system confirmed its successful alignment with all predefined requirements (Requirements OTA.1 through OTA.5), effectively ensuring the system's ability to safely overtake an obstacle in front, under various driving conditions.

16.7 Platooning

The Platooning driving function should assure that a convoy-like formation is created and kept. In Figure 16.10 the ego vehicle is located on the middle lane and driving. Behind it is another vehicle and behind that another. All the vehicles keep the same distance from each other which is within a predefined range. In contrast to the ACC driving function, the last member of the platoon should immediately react to breaking by the EGO vehicle. The ACC driving function only reacts to the vehicle directly in front of itself. Thus communication between members is a requirement.

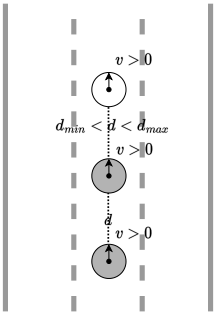


Figure 16.10: Platooning scenario

The framework of TurtleCar is suited for the implementation of platooning. It is important to note that within this project group, only a concept for platooning is developed. The project group's focus layed elsewhere and thus no requirements for a controller were elecitated as only the basic communication infrastructure was planned. Research shows that implementation of the concept is feasible, and the needed communication infrastructure and interface are included in the project. However, a platooning controller is not developed due to time constraints. Additionally, current state-of-the-art platooning implementations are presented and some of the methods used are adapted to enable seamless integration for possible future project groups.

For this driving function no requirements were elicitated. The project group's focus was placed elsewhere and thus it was not worked on initially. Since one of the scenarios described platooning as a desired driving function at least a basic communication infrastructure is provided.

State of the Art

Platooning or Cooperative Adaptive Cruise Control (CACC) describes the intercommunication between Vehicle-to-Everything (V2X) [65]. This communication

model is also used by the AuNa (Autonomous Navigation System Simulator) framework [29]. The presented concept is based on this with additional extensions. The CACC controller includes the algorithm for the position calculation including vehicle dimensions and distances. However, it should also be mentioned that this is a simulative environment, where an extra server or client is also used for communication, which the TurtleCar-Core does not provide. It is also used with callback functions that follow a time interval [29], which is a functionality provided by ROS.

There are also CACC Platform implementations that follow a practice-orientated approach, as shown by textscPohlmann et al. [65]. Although a simulator is also used here, the paper refers primarily to the hardware and software implementation of the ROS and Microros components, which fulfill a real-time part of the system. In conclusion, this research shows the feasibility of creating a working platooning implementation on the TB platform using the mentioned technologies.

Furthermore, the master's thesis by Philipp Fritz Jaß confirms the presented concept in a similar form. However, that platooning implementation uses a different platform, which makes it difficult to establish a direct connection. The architectural approaches are the same but cannot be compared directly with TurtleCar-Core [34].

Interface defintion

This subsection presents a top-down view of the interface and what it needs to provide to be capable of platooning. The interface is defined as a hook-in module, which means that the functionality can be activated or deactivated as required. This way all other driving functions remain uninfluenced by creating the interface for platooning. The hook-in module does not depend on any functions in the TurtleCar-Core framework. It also offers some configuration options: A configuration must include which sensor data should be shared, which driving functions are to be activated, and how many vehicles will communicate with each other. Different communication paradigms are realized by the definition of the communication interface. As an example, it might make sense for a member of a platoon to notify other members about a collision risk. A member who has already passed such risk does not have to be notified, however.

Since the basic functionality of TurtleCar-Core is used, platooning can be viewed similarly to a sensor: various events transmit data which is analyzed by a sensor evaluator. Then the relevant data is written into the model. This data can then be reacted to accordingly by a controller, which implements the platooning driving function. The parts affected by platooning can be seen in Figure 16.11.

Platooning topics are defined externally and can be subscribed to. Events are defined in advance accordingly. Each event can be uniquely identified, e. g. the Collision Risk, Collision, Obstacle detected event exists below Figure 16.12. Each of these events is sent as a message, including the respective identifier of the vehicle as well as its current coordinates. A map can thus be built up, which allows making statements about the entire geometric space currently measured.

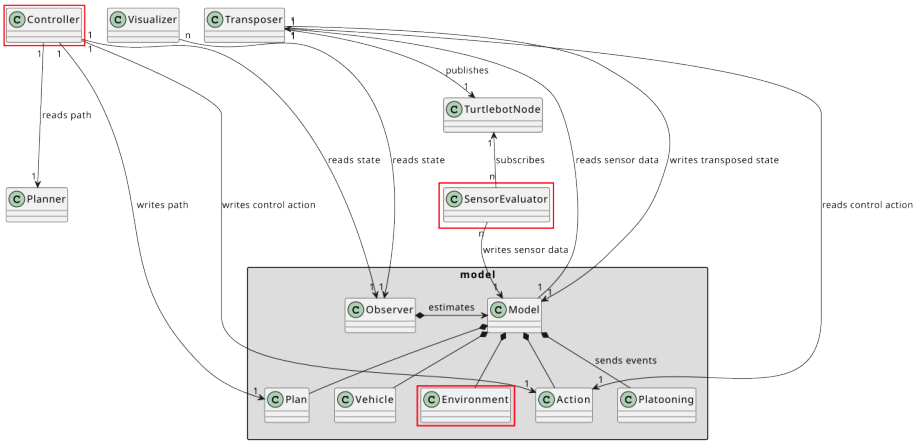


Figure 16.11: Parts of TurtleCar-Core used by the platooning module

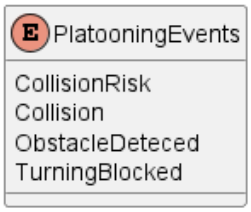


Figure 16.12: The defined events for the Platooning node

Communication

The platooning messages are the basis of all communication within the network and allow the broadcasting of the positions of platoon members regularly. The message format includes an identifier and the necessary information about the publishing vehicle's position.

Communication within a platoon is possible in different ways. A general case is depicted in Figure 16.13. By nature of the ROS network, all communication at the lowest level is done via broadcasting. That way all events received within a platoon are shared via a single topic. Thus the implemented controllers and evaluators of the respective vehicles that handle platooning have to use communication different from a broadcast. In Figure 16.13 there are however also topics within the namespace of each participant. Sending a message to such a topic represents a direct communication between two participants. The messages are still broadcast, but only received in a single namespace, which all participants could still receive. However, the encapsulation using the namespace concept indicates the usage as non-broadcast communication.

A header message with the necessary information about the communication participants thus has to be defined. This approach is seen in Table 16.4.

For bidirectional communication in platooning mode, ROS services can be utilized. As services, they are designed to provide a result and can only be hosted by one peer at a time, allowing the representation of a decentral communication

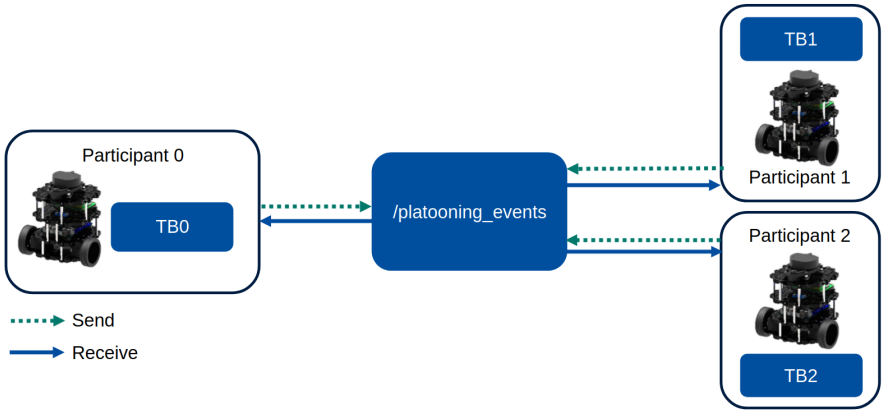


Figure 16.13: Communication via ROS

Table 16.4: Platooning Header

Parameter	Type	Description	Notes
identifier	string	Identifier of sending vehicle	
destinations	string[]	Identifiers of vehicles that should be reached	Should be empty for broadcast

paradigm.

Events use custom messages from the new platooning messages interface. These messages enable different communication paradigms by using the platooning header or being offered as services. In Table 16.5 an exemplary definition of such a custom message is given. This is done similarly to how the previously mentioned AuNa defines messages for communication [29].

Table 16.5: Definition of ObstacleDetected

Parameter	Type	Description	Notes
header	platooning_header	Header with communication information	-
obstacle_type	unit8	Information about the type of obstacle	-
obstacle_vertices	geometry_msgs/Point[]	Vertices of the detected obstacle	In global coordinates

Implementation

In order to provide a proof-of-concept for the communication infrastructure necessary for platooning, the use case for sharing information about obstacles was chosen. Members of a platoon should be able to update their respective environment models using information gathered by other members to include obstacles that they can not observe using their own sensors. This may be the case when the sight to the obstacle is obstructed or the obstacle is outside of the sensors' range. This proof of concept assumes that all vehicles share a common reference frame. This can be easily created by using global coordinates of a simulated environment. In real life, a synchronisation mechanism would have to be implemented, which was out of scope for this project.

For this, a common topic `/platoon_events` is used by all platoon members. Each member regularly broadcasts String messages to that topic. As they are a proof of concept, the messages are a simplified structure compared to the one defined above. Each message contains a String representation of a JSON object. The object can represent several types of messages and has the following structure:

Table 16.6: Definition of a PlatoonEvents message

Name	Type	Description
type	String	Either `ObstacleDetected` or `TurtleCarPosition`
data	JSON Object	Information depending on the type

The `ObstacleDetected` message contains information about the position, width and length of an obstacle. Each platoon member regularly broadcasts messages for all its known obstacles. The structure of the `data` object of this message can be seen in Table 16.7.

Table 16.7: Definition of the data object of an ObstacleDetected PlatoonEvent message

Name	Type	Description
ID	Integer	The ID of the obstacle as determined by the sensing vehicle
obstacle_type	String	RoadSign, StaticObstacle or TurtleBotPlate
x	Float	The x coordinate of the obstacle
y	Float	The y coordinate of the obstacle
width	Float	The width of the obstacle
length	Float	The length of the obstacle

This is limited to obstacles that have been perceived by its own sensors to avoid sending duplicates. If a platoon member received such a message, it integrates the obstacle into its own environment model. To avoid duplicating an obstacle that it already knows, it checks whether it knows the ID of the obstacle or if the distance between the received obstacle and any known obstacle is smaller than a threshold value.

The `TurtleCarPosition` message contains information about the position of a platoon member. Each platoon member regularly broadcasts its own position

using this message. The structure of the `data` object of this message can be seen in Table 16.8. The receiver integrates the position into its environment model in the same way as for other obstacles.

Table 16.8: Definition of the data object of an TurtleCarPosition PlatoonEvent message

Name	Type	Description
ID	Integer	The ID of the obstacle as determined by the sensing vehicle
obstacle_type	String	RoadSign, StaticObstacle or TurtleBotPlate
x	Float	The x coordinate of the obstacle
y	Float	The y coordinate of the obstacle
width	Float	The width of the obstacle
length	Float	The length of the obstacle

The resulting environment models when sharing obstacle information between platoon members can be seen in Figure 16.14. The top half of the image shows the simulation with three vehicles, which are members of the same platoon. Vehicle 1 can see vehicle 2, but not the obstacle and vehicle 3. Vehicle 2 can see vehicle 1 and the obstacle, but not vehicle 3. Vehicle 3 can only see the obstacle. The bottom half shows the visual representation of the environment models of each vehicle. It can be seen that all vehicles know the position of all other vehicles as well as the obstacle.

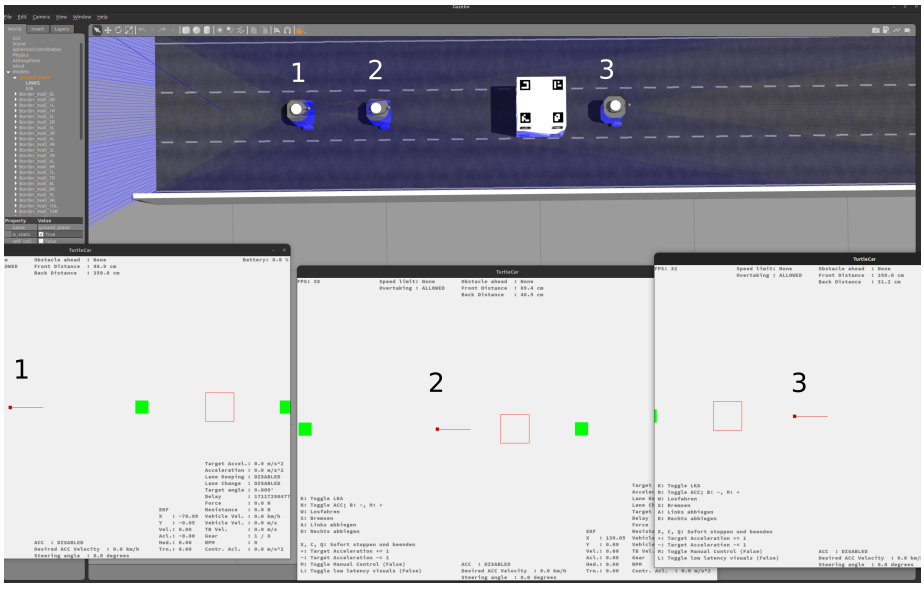


Figure 16.14: A simulated environment with three platoon members sharing their positions and obstacle information.

16.8 Constraints on Driving Function

Multiple driving functions are generally unable to be activated at the same time, because of conflicting control inputs. For example, the LKA wants to keep to the center of its current lane, while a LCA wants to move from the current lane to another. This section describes constraints on the simultaneous activation of driving functions. Since there are conceptual differences between an approach based on MPC and approaches that aren't based on MPC a distinction between these is made.

16.8.1 Classic Approach

Based on the implementations for different driving function, it is apparent which control inputs a given controller calculates. These inputs are then used to control the vehicle. This information provides a way to establish which driving functions can't be active simultaneously. Using this approach first results can be seen in Table 16.9. The + symbolizes that the functions can be used together, while the - symbolizes that they cannot. The black boxes means that they are the same function.

Table 16.9: Driving functions that can be active simultaneously

Driving Function	LKA	ACC	LCA	CAS	Overtaking
LKA		+	-	-	-
ACC	+		+	+	-
LCA	-	+		-	-
CAS	-	+	-		-
Overtaking	-	-	-	-	

It is apparent, that most driving functions can't be used at the same time. An exception to this is the ACC which can be used in conjunction with one of the other functions that don't generate an input for accelaration.

However, this does not show the whole picture, as some driving functions behave similarly to others, like Overtaking and Obstacle Avoidance. At their core, these functions make the vehicle change the lane to drive past another object. In the case of Overtaking this object is also moving, while for Obstacle Avoidance it remains stationary. Additionally, both functions make the vehicle change back to its original lane after avoiding an obstacle. This behavior can be compared to using the LCA twice.

The fully autonomous vehicle has to drive in a safe way, which implicitly establishes a hierarchy for the driving functions. The vehicle should use the Obstacle Avoidance or Overtaking driving function if there is an impending collision with another object, rather than using the ACC or LKA.

Platooning behaves a bit differently here because the leader of the platoon (the ego vehicle) is set to act using possibly all other driving functions, while the platoon members should just mimic the ego vehicle's actions and not use the functions themselves. This way the members aren't involved in planning actions, but just in the perception of fellow members and looking out for possible collisions with non-platoon objects.

16.8.2 Model Predictive Control Approach

The MPC consists of two layers. First, there is a path planner and second, there is a path follower. Every function can be implemented using either layer, with different degrees of complexity. This changes how driving functions interact with each other. For example, the Obstacle Avoidance function can define the constraints for a trajectory to forbid „driving through“ an obstacle. This forces the path follower to drive around the obstacle automatically if the trajectory is still feasible. The other option is to re-plan the trajectory around the obstacle. This would mean that the main function of the Obstacle Avoidance is done in the planning layer. These ideas are the same for Overtaking.

Having more than one MPC-based controller run at the same time would mean following multiple trajectories. This may lead to conflicting goals of the participating controllers and is therefore discouraged. From the driving functions developed within this project group, only one of the LCA, LKA or OTA can be used at a time. Since the driving functions using MPC are only used to control the steering angle, it is possible to combine either of the MPC-based controllers with the ACC which only controls the acceleration and is not MPC-based. It is, however, necessary to update the trajectories and constraints of the MPC-based controller regularly as the acceleration is changed by the ACC. It is important in these situations to ensure that the acceleration does not change to frequently, since the model prediction will be differing from reality if it assumes a constant acceleration. This can be mitigated by predicting the acceleration set by the ACC.

Another caveat for the ACC if approached in an MPC context, is that it requires constant monitoring of the lead vehicle. The acceleration of the ego vehicle based on the lead's vehicle speed and acceleration as well as the distance between the two should be able to change often. This results in a constant updates for the MPC, which is not something that should be done in the MPC approach.

One overall solution to the problems described would be to design one trajectory that combines the functionality of the ACC and the other active driving functions. This can be sed as input for an MPC which can generate optimal control signals for both acceleration and steering and therefore has complete knowledge of the vehicle input and state for each time step, resulting in a better prediction.

Chapter 17

Scenarios

17.1 Main Scenario

The „main“ scenario is designed to adhere to German traffic laws, specifically focusing on passenger cars (PKW) and excluding commercial trucks (LKW). The traffic scenarios simulated within this framework are bound by speed limits of 80 $^{km}/_{h}$ or areas with unrestricted speed, reflecting the typical regulations found on German autobahns. Also a sign for a restriction for overtaking is used. Given the constraints of the highway scenario, many of the aforementioned dangerous driving behaviors are not directly applicable; however, the emphasis on speed regulation adherence and lane discipline remains paramount. Within this scenario, all vehicles strictly follow the German traffic regulations pertinent to passenger cars, with particular attention to speed limits and overtaking rules. This not only ensures the realism of the simulation but also enhances the safety and reliability of the autonomous driving algorithms under test.

Two primary causes of dangerous situations identified for in-depth analysis within this framework are unexpected lane changes and sudden braking. These events are of particular interest due to their high relevance on highways and their potential to cause significant accidents. By simulating these events within a legal and realistic framework, the project groups aims to rigorously test the autonomous vehicle's response mechanisms. The vehicles' ability to adhere to traffic laws while effectively handling sudden changes in the traffic environment is critical for the development of safe and reliable autonomous driving technologies. Furthermore, this scenario facilitates the examination of autonomous vehicles' behavior in both regulated speed zones and on sections of autobahns without speed limits. This dual approach allows for a comprehensive assessment of the vehicles' speed management, overtaking strategies and general adherence to traffic regulations, providing valuable insights into their operational safety and efficiency.

In conclusion, the main scenario represents a significant step towards the development of autonomous vehicles that are not only technologically advanced but also fully compliant with traffic laws. By grounding the simulations in the realities of German autobahn driving, the project groups aims to pave the way for the safe integration of autonomous vehicles into public roadways, ensuring that they can coexist harmoniously with human drivers and adhere to the

established rules of the road.

17.2 Rogue Actor

A rogue actor represents a vehicle that does not adhere to the expected traffic rules or behaviors, potentially creating dangerous situations. In the context of the scenarios, 'Rouge Actor Scenarios' involve dangerous situations created by a rogue actor on the highway, where interactions are limited. These scenarios are designed to test the autonomous driving functions of the vehicles in handling sudden, unexpected behaviors from other vehicles on the road.

In a paper by KEONHEE&AKIRA, an analysis of the most frequently occurring dangerous driving events is conducted based on recorded data from over 3,600 study participants [56]. The analysis reveals that the most common dangerous road situations include stop sign violations, followed by interference with pedestrians, traffic light violations, and speeding. Other sources report similar findings, highlighting these violation types as frequent causes of accidents. Additionally, other factors contributing to accidents and dangerous road situations, such as tailgating, reckless driving, unsafe lane changes, and driving under the influence are listed [89]. However, given that the scenarios are confined to the environment of a highway, the range of dangerous situations that can be simulated is limited. For instance, the scenarios assume that the only present road users are vehicles and that there are no traffic lights to consider. These limitations narrow the scope of rogue actor scenarios that can be simulated within the environment. Two causes of dangerous situations, become particularly relevant for testing of the autonomous vehicles: (1) unexpected lane changes and (2) unexpected braking, both of which are replicable in the simulated environment.

17.2.1 Scenario 1: Unexpected Lane Change

Description: This scenario simulates a situation where a rogue actor performs an unexpected lane change in front of the ego vehicle. The scenario aims to test the vehicle's ability to detect the dangerous lane change quickly and to adjust its driving path accordingly to maintain a safe distance. The scenario can be seen in Figure 17.1a.

How to replicate this scenario:

1. Initialize the simulation environment on a straight road with two vehicles starting side by side.

2. Define which vehicle is controlled autonomously and which one is the rogue actor.

3. Start both vehicles at the same time, with equal constant acceleration.

4. Manually increase the acceleration of the Rouge Actor, accelerate in front of the autonomous ego vehicle and switch to its lane a short distance in front of it.

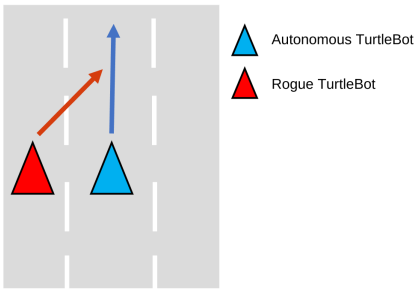
5. Monitor the autonomous vehicle's response to the unexpected lane change.

17.2.2 Scenario 2: Unexpected Braking

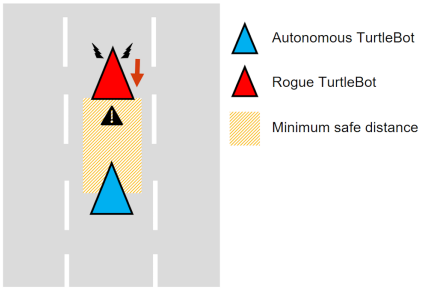
Description: In this scenario, a rogue actor starts at a safe distance in front of the autonomous ego vehicle, and after driving for a moment, abruptly brakes to a lower speed. This scenario tests whether the vehicle is able to react quickly enough and keep a safe distance to the unexpectedly braking vehicle. The scenario is depicted in Figure 17.1b.

How to replicate this scenario:

1. Initialize the simulation environment on a straight road with two vehicles, the Rouge Actor starting two vehicle lengths in front of the autonomous one.
2. Start both vehicles with the same acceleration.
3. Briefly brake with the Rouge Actor to decrease the distance to the vehicle following it.
4. Monitor the autonomous ego vehicle's reaction in this situation, is it able to restore a safe following distance?



(a) Example of an unsafe lane change situation.



(b) Example of a vehicle suddenly braking.

Chapter 18

Organization

This section describes everything related to the internal organization of the project group. A more detailed description on how the product vision will be achieved is provided here.

18.1 Milestones and Timeline

In order to reach the project group's goal, the following four milestones as listed in Table 18.1 were defined in the beginning.

Table 18.1: Planned milestones

Milestone	**Start date**	**End date**
MS 1: Lane keeping assistant and fundamental architecture	05.05.2023	08.09.2023
MS 2: Adaptive cruise control and basic testbed features	09.09.2023	28.09.2023
MS 3: Autonomy Features, Robot Vision	29.09.2023	22.12.2023
MS 4: Rogue actor and platooning	23.12.2023	07.03.2023

Furthermore, a more detailed time schedule depicted in Figure 18.1 was offered initially. The thick vertical lines depict the end of a milestone. Additionally, the epic can be grouped together as follows: driving functions (green), test framework (orange), obstacle avoidance (purple), platooning (blue), documentation (yellow), and higher-level (grey).

Later, the milestones and the timeline were adjusted regarding the content and duration as agreed upon with interested parties. This had to be done due to unexpected drawbacks like cases of illness and a member resigning. These changes are depicted in Figure 18.2.

This rework results in the following new timeline in Table 18.2.

In regard to content, the scope of sensor fusion was reduced in a way that the product vision would still be fulfilled. A theoretical approach was promised but not an implementation. Additionally, the scope of platooning was reduced since it was planned as an optional goal. Hence, the definition and implementation of a fundamental architecture for platooning was promised to enable a simple starting point in implementing platooning for future works.

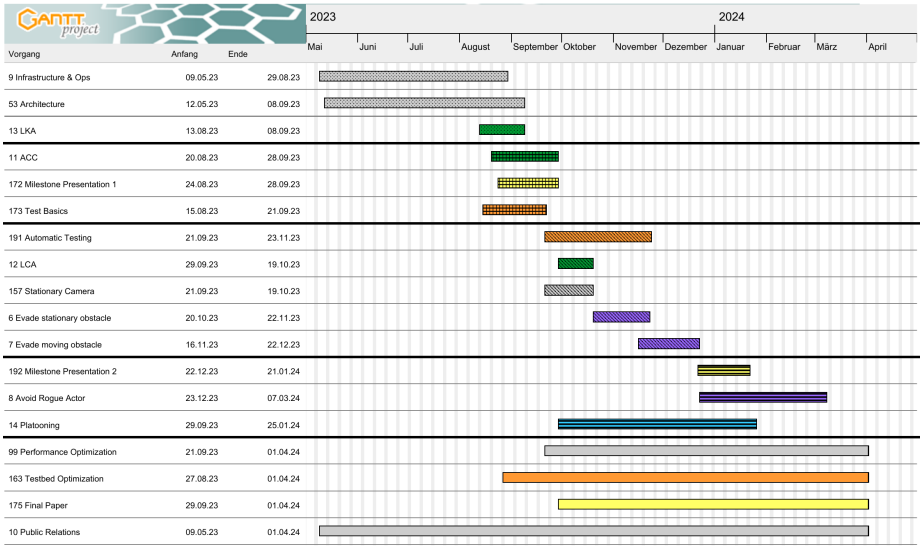


Figure 18.1: Gantt chart of epics

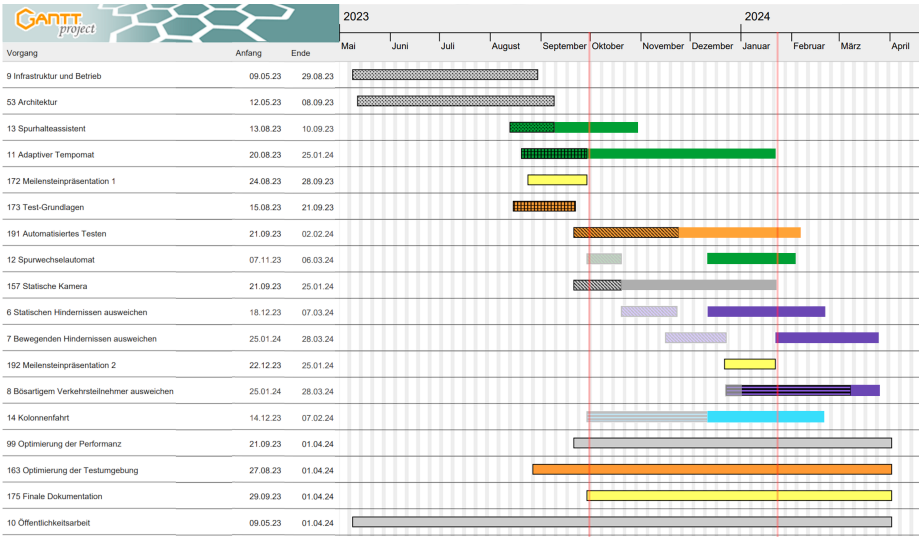


Figure 18.2: Rework of the epics' gantt chart

18.2 Sprint-flow

The previously defined milestones will be archived in an agile way using the scrum process [77]. A sprint lasts three weeks and consists of the following aspects:

- **Feature-Planning (FP)**
  In this phase Product Owner (PO) and Business Engineer (BE) and all interested parties consider which features should be developed in the future to reach the milestones. The features are recorded in Jira. Tickets are

Table 18.2: Rework of the planned milestones

Milestone	Start date	End date
MS 1: Lane keeping assistant and fundamental architecture	05.05.2023	08.09.2023
MS 2: Adaptive cruise control and basic testbed features	09.09.2023	28.09.2023
MS 3: Autonomy Features, Robot Vision	29.09.2023	15.02.2024
MS 4: Rogue actor and platooning	16.02.2024	21.03.2023

created that contain the needed requirements.

- **Implementation**
  During this period, the tickets are processed, documentation is written, and reviews are performed by others so that they can finally be merged.

- **Review**
  The goal of the review is to bring all stakeholders up to date. It should be mentioned which goals have been achieved and the progress should be presented.

- **Retrospective**
  The team sits down internally at the retrospective at the end of the sprint and draws a summary. The focus is on filtering out problems, exploring possible solutions and citing positive aspects.

During the sprint, a weekly serves as an exchange with the stakeholders by giving a quick presentation of last week's progress. Internally, meetings are scheduled twice a week. Once every sprint, a refinement of the backlog is planned which is done to facilitate the feature planning and refine Jira tickets to enable faster sprint plannings.

18.3 Sprint Workflow

To assure that all members follow the same workflow regarding the arising tasks during a sprint, the following well-defined workflows have been agreed upon. For example, every sprint follows a specific workflow. An overview is given in Figure 18.3, where every colored step (except the „Sprint planning“) resembles one column in a Jira Sprint Board, as it is shown in Figure 18.4. First, the sprint has to be planned. Every ticket in the sprint is then assigned to one or more people. When they have finished processing the ticket, it goes into review and finally into acceptance by the PO or BE.

Since some of these steps are complex in nature, it is important to clearly define their respective workflows. This is done by the following diagrams, with continued usage of the color coding as seen above. The „Sprint planning“ workflow is described in Figure 18.5. The activity „To-Do“ is empty, as this step only consists of waiting for any ticket-related work to start, thus requiring no well-defined workflow. The workflow described in Figure 18.6 shows how tickets that are in progress should be worked on. The workflow for „To Review“ and

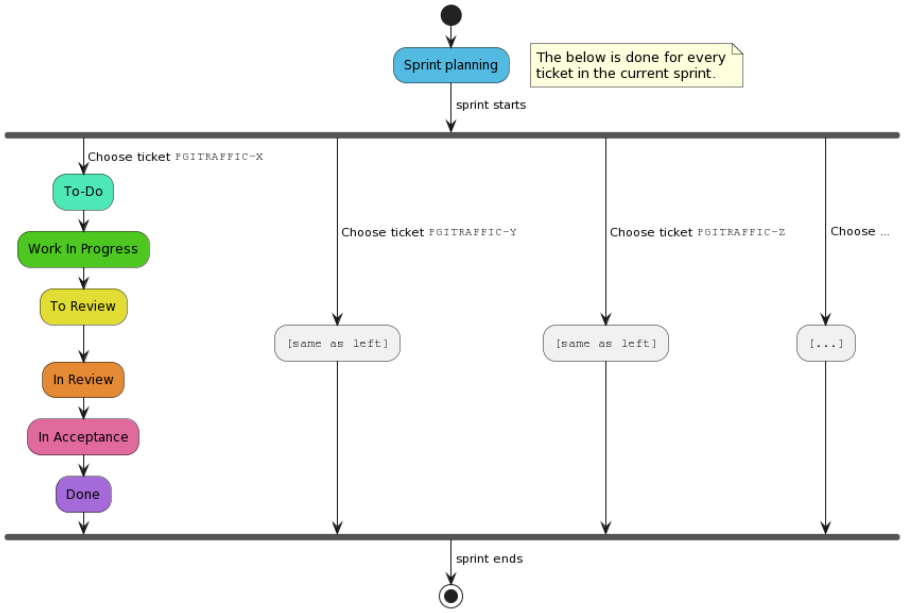


Figure 18.3: Overview of how the work on items in a sprint is done.

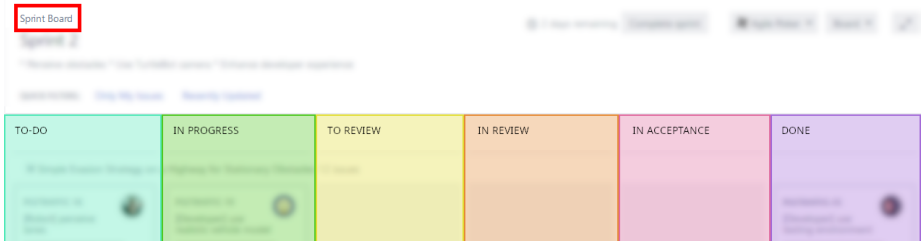


Figure 18.4: The states of an issue as represented in the Jira board.

„In Review“ is shown in Figure 18.7 and the workflow for finalizing a ticket is described in Figure 18.8.

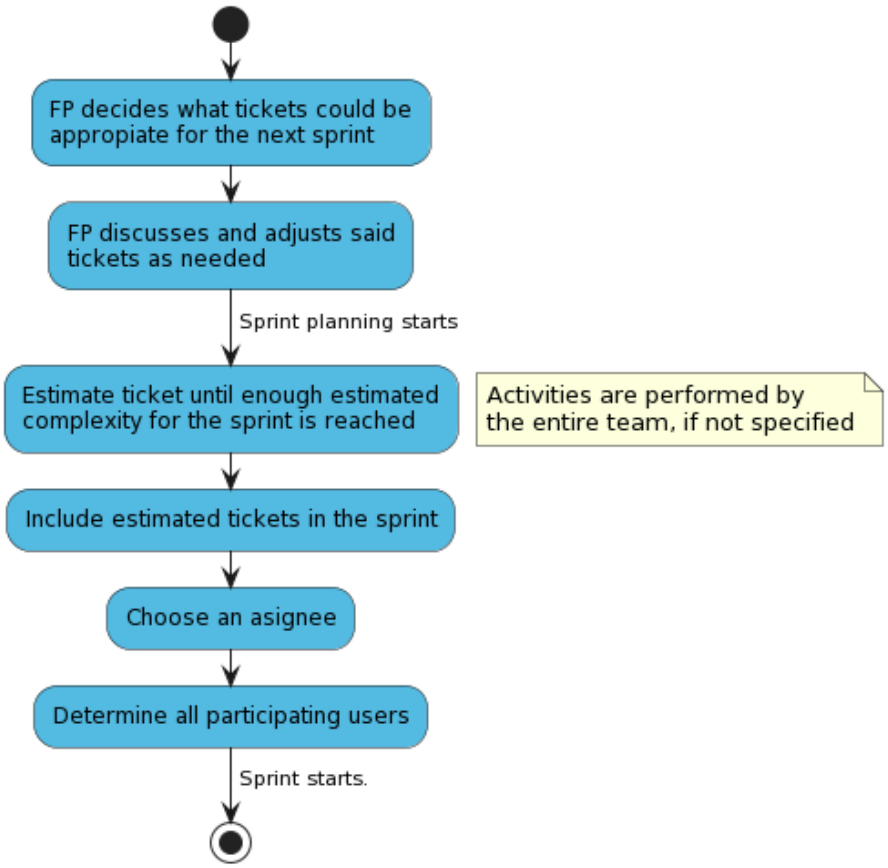


Figure 18.5: Sprint planning workflow

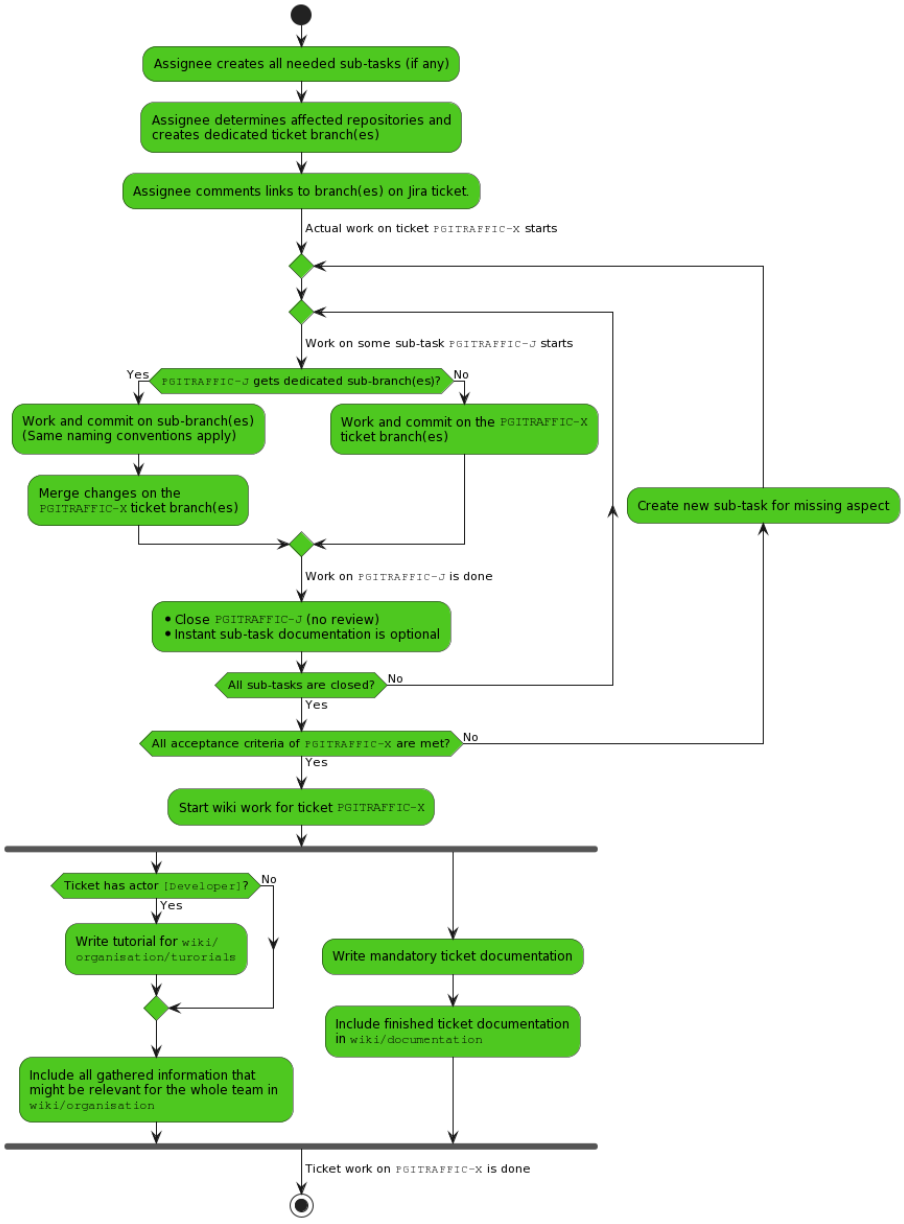


Figure 18.6: Work in Progress workflow

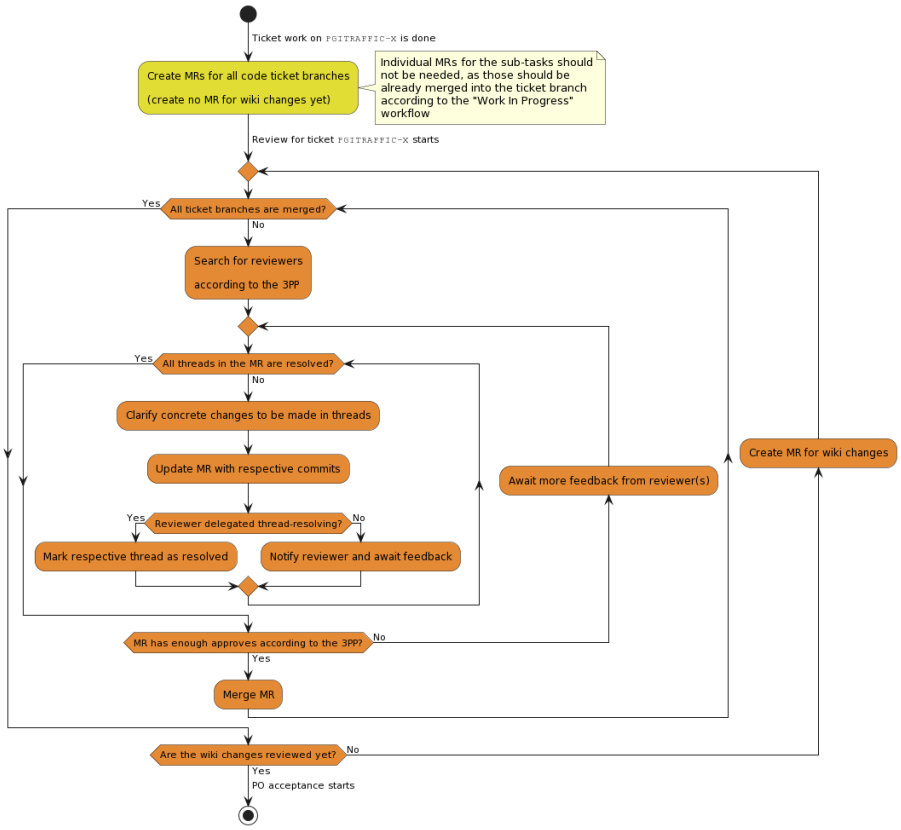


Figure 18.7: To Review and In Review workflow

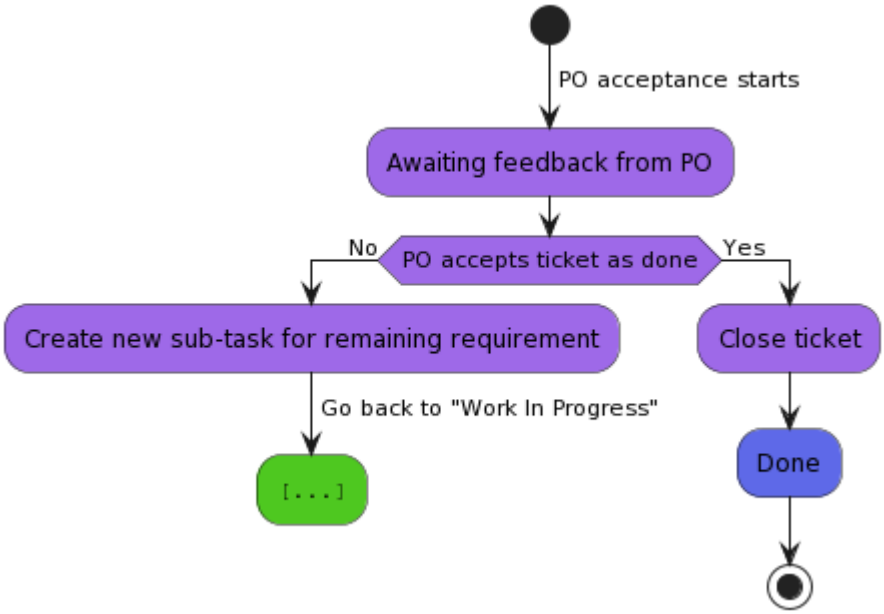


Figure 18.8: In Acceptance workflow

18.4 Defintion of Done

A ticket is considered done if the following requirements are fulfilled:

- Functionality implemented
- Reproducibly tested
- Documented
- At least three persons were involved in implementation and review, at least one of them is only a reviewer
- All acceptance criteria are met
- Accepted by PO or BE

18.5 Roles

The project group consists of eleven students. Each member is a developer, but some also fulfill different roles or focus on certain topics. These roles with the member's names inside this project are listed below.

Scrum Master (Carl Schneiders) : The Scrum Master ensures conformity to Scrum practices and maintains the team's processes. Carl is responsible for removing obstacles in the process and organizing retrospectives.

Product Owner (Marie Marken) : As the Product Owner, Marie creates and maintains the product vision in consultation with the group and other interested parties. She communicates with BTC ES, Foundations and Applications of Systems of Cyber-Physical-Systems and Distributed Control in Interconnected Systems. Additionally, she maintains the backlog, organizes feature planning, and leads sprint planning and sprint review.

Business Engineer (Lasse Heckelmann) : The Business Engineer supports the Product Owner in her tasks and keeps track of the project. Lasse also provide a point of contact for specialized questions.

Documentation Steward (Nellson Eilers) : Nellson is responsible for keeping track of the internal wiki, ensuring that everyone documents their work, and adhering to conventions regarding documentation.

Infrastructure (Malte Grave) : Malte maintains the server infrastructure, ensuring that everyone can work smoothly. Additionally, he offers technical support as needed.

Code Steward (Jan-Magnus Monenschein) : Jan-Magnus ensures that the code quality meets the desired level by specifying rules and principles for working on the code base. He also assists with Continuous Integration/Continuous Deployment (CI/CD) and configuring development tools.

Technical Lead (Simon Struck) : Simon has an oversight of the entire system, responsible for the creation and management of data transfer protocols. They also evaluate technical feasibility of various aspects of the project.

Quality Analysis (Filip Wojciak) : Filip Wojciak ensures product functionality by testing it thoroughly. He is tasked with testing the product and ensuring it fulfills all requirements.

Developer (Julia Debkowski) : Julia assists with software development and keeps track of the project's progress.

Software Architect (Stefan Gerber) : Stefan maintains the architecture of the software and serves as a contact for architectural questions.

PR work (Paulina Kowalska) : Paulina is responsible for public relations work and planning events related to the project.

18.6 Tools

For easier collaboration, using a few tools proved to be essential. Below, some of these tools are presented.

Jira Jira is a popular project management and issue tracking tool developed by Atlassian. Jira helps to manage tasks efficiently, maintain transparency, adapt to different project methodologies and collaborate effectively.
Jira allows teams to create, track, and manage issues, tasks, bugs, and user stories. This helps in maintaining a clear and organized list of work items, making it easier to prioritize and address them. Also, Jira supports agile methodologies like Scrum. It provides features such as sprint planning, backlog management, and burndown charts to facilitate agile processes.
Furthermore, Jira is highly customizable. This enables the ability to have custom workflows, issue types, and fields to tailor it to a project's specific needs.
Another important aspect is that Jira can integrate with a wide range of tools, including source code repositories (i. e. Gitlab), CI/CD pipelines and more.

Discord Discord is used for communication within the team. A custom bot called Hugo is used, which partially automates processes. Particularly, he reminds the group of the internal weekly deadline, helps with the estimation process of user stories and can be used to list current merge requests and their review status.

Gitlab Versioning is essential. Gitlab is used for this purpose. Repositories for the following projects exist:

- Internal wiki
- Website
- Public Relations
- Project Report
- Server Configuration
- TurtleCar

Google Calendar To keep track of important dates Google Calendar is used. Here all appointments as well as vacations are entered.

Etherpad Etherpad is used to share notes and to keep the agenda for meetings.

Chapter 19

Public Relations

In this section, the presentation to the public will be addressed. This will cover tasks performed during participation in events like the FleiWa, as well as the management of the project group's online presence, including a website and Instagram account.

19.1 Quartierstag

Figure 19.1: Presentation at the Quartierstag

At the „Alte Fleiwa“ neighborhood, as part of its 100^{th}-anniversary celebration the „Quartierstag“ was held. Here a first major milestone, the LKA, was

presented. This can be seen in Figure 19.1. During this event, local businesses, research institutions, organizations, and municipal offices provided insights into their work. More information can be found on the following link: `https://quartierstag.de/` On behalf of the University and BTC-ES, current findings were presented, a live demonstration was offered, a poster as seen in Figure 19.2 was created and the opportunity to examine hardware and software, including the Visualizer was given.

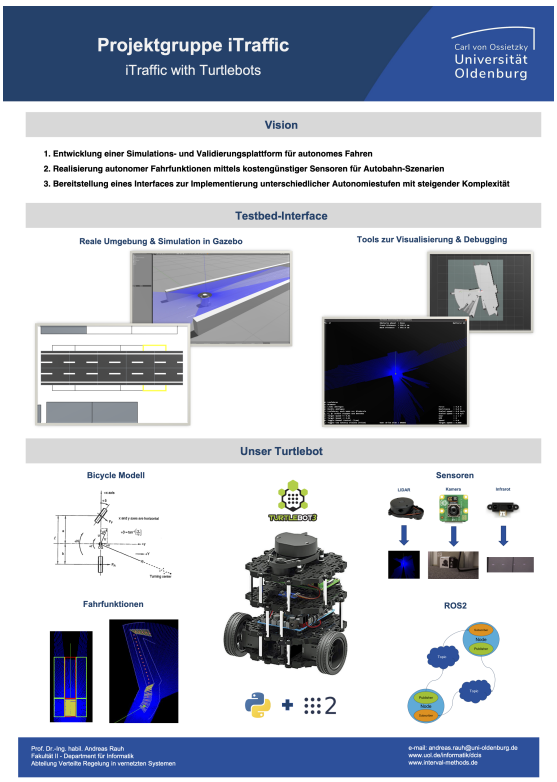


Figure 19.2: Overview of the poster for the Quartierstag.

19.2 Website

In today's world, it is of paramount importance to establish an online presence. To this end, a digital presence was created. First, an Instagram account exists, which will be actively curated in the near future. What is already accessible by the public is the website. The project's website displays the most important information for the public. Its public domain is `https://itraffic-uol.de/`.

The website is a good way to document the progress being made over time and to also show it to stakeholders. The content should primarily address the goals of the project group. Progress should be documented as well as challenges to avoid or not to repeat possible mistakes. The team represents the basic building block of the project group and is therefore presented. This way, even strangers who have nothing to do with the project group can build a good understanding of it.

19.2.1 Dependencies

The following tools are used to create the website:

- Jekyll (Static Site Generator)
- Minimal Mistakes Theme for Jekyll
- Ruby's bundler gem in order to manage the projects dependencies
- Gitlab CI/CD for building and deploying the site automatically

19.2.2 Content Review Policy

Since the content affects everyone and appears online, changes should be approved by everyone in advance. Joint reviews are mandatory.

19.3 Email

The teams public email address is: team@itraffic-uol.de

19.4 Instagram

The project group's Instagram channel can be found here: `https://www.instagram.com/pg_itraffic/`

The Instagram channel is a bit more informal and is intended to represent the project group away from the achievement of goals. For this purpose, insights into meetings but also social events can be shared. Regularity to post is secondary.

Chapter 20

Outlook

In this section an overview is given on where the project group saw great potential for further work. These ideas can be seen as having an extent ranging from a smaller university course over a thesis up to another year long project group. Some of these ideas have been considered during work on related tasks within the project group. These were deemed to be too large in scope and not relevant enough for the current goals, which is why they are discussed here.

This outlook is split in three parts. Each of corresponding to a product the project group developed. Fist an outlook on possible driving function are considered, second an outlook for TurtleCar-Test is given and lastly the possible future regarding TurtleCar-Core is discussed.

20.1 Driving Functions

Controller for Platooning The first obvious starting point for continuing to develop driving funtion in the context of the scenario used by the project group is the implementation of the platooning driving function. Due to time constraints only the communication infrastructure has been implemented as seen in Section 16.7. Using these basic blocks the implementation should be relatively straight-forward as all necessary sensor information is already provided. Here the next steps can include depending on the desired complexity:

- Implement a controller that is able to send commands to all vehicles of a platoon.
- A method to enter, leave or found a platoon.
- Negotiate which vehicle is the leader of the platoon.
- Identifiy safe ways to handle emergencies within the platoon, (i. e. the failure of a vehicle)

With all these steps to consider this driving function is deemed to be of lower complexity in contrast to other features considered in this outlook, because only a single new controller has to be develop. Such a development process for any controller is exactly what the project group envisioned when designing TurtleCar-Core.

Developing other controller Developing any other controller for any possible driving function is also possible using the aforementioned development process. All information about the environment is provided by the observer and can be accessed by the controller under development. Such driving functions have not been explicitly considered by the project group. One could image however manoevers such as merging into another lane in a high traffic situation or create the emergency lane on the highway in case of a traffic jam.

Different Scenarios Other ideas brought forth during the discussions of new driving function require a lot more work, because they are only applicable in environments different from the German autobahn, which was the project group's focus. One such idea is a slower paced environment with more precise manoevers. Such an environment could for example be a parking lot.

Parking Lot A parking lot is generally a low-speed environment and the main feature is parking. Since the TBs work with local knowledge only, searching and evaluating if a parking spot is occupied or if it is large enough for the vehicle in question. Then a manouver for different parking strategies has to be chosen depending on size of the parking spot, the position of the vehicle relative to the parking spot and the orientation of the parking spot. Since there are many different types of parking spots and often other drivers don't park perfectly inside their spot, there are many challanges to consider. One such challange is incoming traffic as there might be other vehicles trying to leave the parking lot or used a different strategy trying to find parking spot. The ego vehicle is supposed to give way if necessary before attempting a to park. Another such challange is the handling of intersections. Here the right of way usually is right before left, which the ego vehicle has to adhere to. This includes estimating if crossing the interection can be done safely with traffic coming from the right. While the function of parking is a more isolated case TurtleCar-Core could be used all assumption based on the autobahn have to be removed and a new environment has to be created.

City Traffic Similarly another idea produced throughout the project group would also introduce the idea of incoming traffic and interesections. This would be moving the environment to one based on city traffic. Here the amount of challanges an autonomous vehicle faces can be sheer endless. There are people close to the street and might try to cross the road in irresponsible ways. The ego vehicle should ensure that still no damage to humans is done to humans. This can possibly be implemenented using the emergency detector (Section 15.1) introduced with the autonomous supervisor. There are many different types of roads in terms of width, lane amount and different types of interesections. Such intersection can have different ways to handle the right of way. One way would be the same „right before left“ rule as in parking lots. Often there are road signs indicating to give way and sometimes there are traffic lights. Detecting road signs is already implemented in TurtleCar-Core, so adding more shouldn't be a huge challange. However, traffic lights could be more challanging, because a computer vision approach would have to be researched and implemented. Lanes might also merge into one or split into multiple. When lanes merge the vehicles have to merge as well. Usually this done using zipper merge, which is a more

standard driving feature and should be easily implementable with the current TurtleCar-Core workflow for developing controllers. Following the correct one or understanding when to change the lane can be difficult. This is especially true when trying to do autonomous navigation, which would be another enormous undertaking in terms of work. In fact it would be considered to be another project entirely. Thus the project groups determined that a realistic depection of traffic in a city is at least the work of a new project group.

Extension to current scenario A smaller scenario which may only be cover work of a thesis is an extention of the current autobahn scenario. The introduction of a merge lane and an exit would require little updates to the already existing environment and its assumption. The development of a zipper merge controller should be the central challange of this extension. Here the vehicle needs to perform a lane change, which is already possible in TurtleCar-Core. However there is the challange that the lane to change to is occupied or will be occupied by the time the vehicle reaches an acceptable speed to merge into the autobahn. Now the vehicle has to decide on a strategy on how to still complete the merge without driving past the merge lane into the emergency lane. It might be possible to slow down a little to let one vehicle pass and merge behind that vehicle or accelerate more to overtake that vehicle first and merge in front of that vehicle. The decision process is then dependant on many factors such as the length of the merge lane, the presence of other vehicles and their speed as well as the ego vehicle ability to accelarate. This work can then be augmented by extensive validation of the controller to ensure the quality of the controller.

20.2 TurtleCar-Test

TurtleCar-Test has been developed in two iteration with the second improving everything the first was missing. During the analysis of features that should be implemented into the second version some feature were brought up that were deemed to be out of scope for that version. Generally these ideas are nice to have, while being unnecessary for the core functionality of validating the developed driving functions.

Traffic Sequence Charts One extension that has already been hinted at in Section 13.1 is the adaption of TSCs. With this the definition of test-cases can be made easier and possibly be used to verify aspects more formally. However, this adaption would require a decent amount of work in terms of translating a TSC to a python specification of a test-case. As already evaluated every aspect described by a TSC is available in TurtleCar-Test. Thus it is possible to map a TSC to a python specification that TurtleCar-Test can read and execute.

Replayability To capture more information about tests a replay and record functionality could be build. This way anyone could record the run of a test and see the results in a reproducable way. Since sensors are generally noisy and don't always measure the exact same values from the same starting position. This could mean that a test failure is caused by inaccurate sensor data and not because of a faulty driving funtion. To verify the cause of such a failure recording of all sensor inputs and control outputs is necessary. Afterwards one

can replay the recording and see exactly what happened during the test run. It should also be possible to only record the sensor inputs and see what kind of control outputs are generated by a driving function when replaying the data. This way a driving function can be developed while having simulated sensor data, without having to start a test run. ROS already has a feature to record and replay called ROS2 Bag. This feature has an API and could be called directly from TurtleCar-Test. Thus only small asjustments have to be made for recording a whole test run. A replay functionality requires more work as TurtleCar-Test does not have this feature in mind. Instead of only being able to read test specifications in the format of python source files, there would be a need to read a ROSBag, which then can be replayed.

Test Generation During the development of driving function all test cases have been written manually based on the requirements derived from the scenarios. This is sufficient to validate that the driving functions developed throughout the project group are performing in a desirable way. To still archieve a much larger range of possible scenarios based on the initial one, there would be a need to employ test generation. Using such generation one would be able to identifiy edge cases and achieve s higher test coverage. As mentioned in Section 12.1 there are some different methods. Randomizing the initial position to a certain extend is possible using TurtleCar-Test, however there is no strategy or result collection and analysis like is necessary in fuzzy testing. Another possible way to implement some kind of test generation is employing model-based testing. TurtleCar-Test could then be given a model of the system under test and its possible environment and generate test cases based on that.

Extending to the real-world Currently TurtleCar-Test is only used with a simulated environemnt. However driving functions were implemented with the intention of using them on the physical TB. This does not allow for automatic validation in the real-world environment. Approaches for real-life testing have already been discussed in Section 12.2.

20.3 TurtleCar-Core

Replacement for the Bicycle Model While the project group has found the KBM to be sufficient to emulate real vehicles in a manner as stated in Chapter 4, there were also some reasons given in Subsection 5.6.2 to replace the model in other scenarios. To do that one could begin by taking a look at the dynamic models discussed in Subsection 5.6.3 First, the usage of a DDM could be evaluated, and extensive quantitative tests conducted, where the performances of the currently implemented controllers are compared. The experiments could be as simple as running the driving functions with the KBM and a new DDM, and comparing the result of these runs with a reference, like an optimal preplanned path. Measurements could be mean error and standard deviations to the reference path.

Furthermore, experiments on slip angle could be conducted to answer questions such as: Is the TB subject to slip, and whether it is significant enough to justify the usage of a dynamic vehicle model? If so, for mobile robots like the TB that take velocity as an input command the development of a velocity-based

dynamic model can be considered. To add to that, exploring the viability of the velocity-based dynamic vehicle model could offer valuable insights into the TurtleCar platform's limitations [60].

Sensors The TB is a platform that is build to be modular and extensible with different sensors. Currently TurtleCar-Core has evaluators for the camera and lidar as well as one for the basic functionality of platooning. This is enough to provide all necessary information for the currently developed controllers. However there are sensors that can still be equipped to the TB. There are infrared sensors that are already installed, but not used since their range and orientation are not useful the project groups scenario. However these sensors would be useful for more precise manoevering since they provide accurate measure for back and side clearances. An example for such a scenario would be the previously mentioned parking lot.

There is also a moisture sensor, which can be used to estimate the moisture of the road. Here the idea is to model roads in different weather conditions, which in turn forces controllers to handle the vehicle differently. In conditions where the road is more slippery there is a need to be more careful with steering the vehicle to avoid for example aquaplaning.

Another available sensor is a GPS sensor. This sensor has not been used in the project group as its accuracy was not high enough for the scenarios. For different use cases like navigation it may be possible to effectivly use GPS.

What is needed to do for all sensors is in addition to creating a sensor evaluator in TurtleCar-Core is making the sensors output availabe via ROS. This means working with low-level interfaces on the TB's RaspberryPi and its devices. This work that has been done by Robotis, the company behind the TB. However even those are not bug-free as the project group found out and fixed for the odometry.

Sensor Fusion See Section 6.3

Different Vehicle Configuration TurtleCar-Core is able to emulate different vehicles as described in Section 5.3. Here only „standard“ passenger cars have been considered. It would be interesting to use configuration for trucks, which have very different dynamics. The much higher mass results in longer accelaration time and higher braking distance. Controllers then have to take this into account, because now there is a different safety distance, which means keeping a bigger distance when using the ACC. Other minor difference in drag should be negligable, because the controllers developed by the project group are robust enough to deal with minor deviations.

Chapter 21

Conclusion

The goal of this project group was to develop and validate driving functions based on the TB. To achieve this, the TurtleCar development and testing platform was created. The development platform, TurtleCar Core, provides developers with a set of pre-defined methods allowing them to create driving functions, such as lane and obstacle detection using multiple sensors, a state estimator for the state of the vehicle, and a ready-to-use implementation of MPC. Using this platform, it is then possible to run the controllers against the model of a real-world car, which is emulated in a scaled-down version on the TB. The testing platform provides a syntax to write tests in an intuitive, human-readable form, which allows building tests for the driving functions implemented in TurtleCar Core without the need to worry about implementation details.

Using these tools, vehicles with different levels of autonomy were implemented for a scenario based on a German Autobahn. This required building several assistance functions: A LKA, an ACC, a LCA and an OTA also applicable for avoiding static obstacles. Each assistance function was validated using a set of test cases derived from the requirements. Using the assistance function as building blocks, two variants of autonomous cars were implemented. The autonomous cars decide on which speed to drive with and which driving functions to activate based on interpretations of the environment. The Passive Driver stays in its lane and drives behind slower cars, while the Maximum Speed Driver overtakes slower obstacles if possible. Similar to the individual driving functions, both autonomous drivers were validated using test cases derived from their requirements.

One specific goal of the project group with regard to the autonomous drivers was their ability to safely react to vehicles that were acting in violation of traffic rules (rogue actors). The tests show that even if a rogue actor creates a situation that is critical to the autonomous vehicle by braking sharply or pulling into their lane, the autonomous drivers counteract this behavious safely by stopping before a collision occurs or by changing the lane to avoid the rogue actor.

During the development of the TurtleCar platform and the driving functions, two focus topics were identified to be crucial for the development process: Providing a framework for MPC and the need to use an EKF to observe the sensor input and compensate for inaccurate measurements. Similarly, in order to provide the testers with an easier and more flexible way to test increasingly complex driving functions, the testing framework underwent a rework during

the project.

By providing the driving functions and using them for the autonomous vehicles, the project group reached its goal of providing various levels of autonomy on the TBs. Their function can be reliably tested using the developed testing platform. The resulting products offer many points where they can be extended in the context of master theses or subsequent project groups.

Acronyms

Adaptive Cruise Control (ACC) . 2, 18, 84, 94, 101–103, 133–141, 147, 151, 153, 159, 160, 183, 185

ArUco (ArUco) This refers to a library for Augmented Reality applications based on OpenCV. 72–76, 122

Degrees of Freedom (DOF) . 34

Differential Drive Model (DDM) . 33, 35, 36, 182

Extended Kalman Filter (EKF) . 41, 42, 45, 49, 76, 102, 185

Kalman Filter (KF) . 41, 45, 46

Kinematic Bicycle Model (KBM) . 25, 33–37, 182

Lane Change Assistant (LCA) . 77, 84, 112, 116, 121, 122, 141–147, 151, 159, 160, 185

Lane Change Assistant Interpreter (LCAI) . 112, 114, 116, 119–122, 151

Lane Keeping Assistant (LKA) . 2, 5, 84, 85, 102, 103, 125–133, 137, 141, 151, 159, 160, 175, 185

Mathematical Vehicle Model (MVM) . 7, 18, 25, 28, 33–36

Model Predictive Control (MPC) . 21, 51, 99, 103–108, 129–132, 144, 151, 159, 160, 185

Overtaking Assistant (OTA) . 103, 121–124, 147–153, 160, 185

Overtaking Assistant Interpreter (OTAI) . 121, 122, 151

Robot Operating System (ROS) This refers to ROS2 (Robot Operating System 2). In version 2, specifications have been changed, which also include concrete implementation changes. The ROS2 version used is ROS2 Humble, which is marked as Long Term Support (LTS). 11, 39, 47, 48, 50, 52–54, 57, 72, 93, 95–97, 99, 100, 154–156, 182, 183

Traffic Sequence Chart (TSC) . 87, 88, 181

TurtleBot (TB) This refers to used TurtleBot's within the project group. 1, 6, 7, 9, 13–19, 21, 25, 28, 34–36, 39–41, 43–45, 47, 48, 52–54, 62, 63, 67, 68, 71, 74, 84, 85, 87, 154, 180, 182, 183, 185, 186

Bibliography

[1] Andrew Alleyne. "A Comparison of Alternative Obstacle Avoidance Strategies for Vehicle Control". en. In: *Vehicle System Dynamics* 27.5-6 (June 1997), pp. 371–392. ISSN: 0042-3114, 1744-5159. DOI: 10.1080/00423119708969337. URL: http://www.tandfonline.com/doi/abs/10.1080/00423119708969337 (visited on 03/17/2024).

[2] AprilRobotics. *AprilTag: a visual fiducial system popular for robotics research.* 2023. URL: https://github.com/AprilRobotics/apriltag (visited on 12/15/2023).

[3] The Zenoh authors. *Integrating ROS2 with Eclipse zenoh.* Apr. 28, 2021. URL: https://zenoh.io/blog/2021-04-28-ros2-integration/.

[4] The Zenoh authors. *Minimizing Discovery Overhead in ROS2.* Mar. 23, 2021. URL: https://zenoh.io/blog/2021-03-23-discovery/.

[5] The Zenoh authors. *What is Zenoh?* Mar. 9, 2024. URL: https://zenoh.io/docs/overview/what-is-zenoh/.

[6] Uli Baumann. *Die Räder stehen fast quer.* July 2022. URL: https://www.auto-motor-und-sport.de/tech-zukunft/zf-easyturn-achse-extrem-lenkung/ (visited on 10/08/2023).

[7] More BHP. *VW MK7 Golf GT 2.0TDI 150 ECU Remap.* URL: https://www.more-bhp.com/volkswagen-golf-remapping/vw-mk7-golf-gt-20tdi-150-ecu-remap.html (visited on 10/08/2023).

[8] *Black Python Formatter GitHub Repository.* Oct. 2023. URL: https://github.com/psf/black (visited on 10/07/2023).

[9] Philipp Borchers et al. *Realtime Controlled Cooperative Autonomous Racing System next generation.* Checked 2023-10-05. Apr. 2018. URL: https://uol.de/f/2/dept/informatik/download/lehre/PGs/PG-RCCARS.pdf (visited on 10/08/2023).

[10] Paolo Bosetti, Mauro Lio, and Andrea Saroldi. "On the human control of vehicles: An experimental study of acceleration". In: *European Transport Research Review* 6 (Sept. 2013). DOI: 10.1007/s12544-013-0120-2.

[11] Nikolai Bräuer et al. *Realtime Controlled Cooperative Autonomous Racing System.* Nov. 2016. URL: https://uol.de/f/2/dept/informatik/download/studium/pg/PG_RCCARS.pdf (visited on 10/05/2023).

[12] *Bremsen.* URL: https://vorschriften.bgn-branchenwissen.de/daten/dguv/70/19.htm (visited on 10/08/2023).

[13] *Bremswege im Vergleich*. Oct. 2019. URL: https://www.adac.de/rund-ums-fahrzeug/autokatalog/autotest/bremswege-vergleich/ (visited on 10/08/2023).

[14] Bundesrepublik Deutschland. *Straßenverkehrsordnung*. 2013. URL: https://www.gesetze-im-internet.de/stvo_2013/ (visited on 10/08/2023).

[15] cfzd. *Ultra-Fast-Lane-Detection*. 2020. URL: https://github.com/cfzd/Ultra-Fast-Lane-Detection.

[16] cfzd. *Ultra-Fast-Lane-Detection-V2*. 2022. URL: https://github.com/cfzd/Ultra-Fast-Lane-Detection-v2.

[17] Rüdiger Cordes. *cw-Werte*. 2022. URL: http://rc.opelgt.org/indexcw.php (visited on 10/08/2023).

[18] Angelo Corsaro et al. "Zenoh: Unifying Communication, Storage and Computation from the Cloud to the Microcontroller". In: DSD 2023 (Sept. 2023).

[19] Werner Damm et al. *Traffic Sequence Charts - From Visualization to Semantics*. Tech. rep. Oct. 2017. URL: http://www.avacs.org/fileadmin/Publikationen/Open/avacs_technical_report_117.pdf.

[20] Celso De La Cruz and Ricardo Carelli. "Dynamic model based formation control and obstacle avoidance of multi-robot systems". en. In: *Robotica* 26.3 (May 2008), pp. 345–356. ISSN: 0263-5747, 1469-8668. DOI: 10.1017/S0263574707004092. URL: https://www.cambridge.org/core/product/identifier/S0263574707004092/type/journal_article (visited on 03/15/2024).

[21] *DQ381 DSG gear ratios?* URL: https://www.golfmk7.com/forums/index.php?threads/dq381-dsg-gear-ratios.360005/ (visited on 04/03/2024).

[22] *DSG Shift Time*. June 2007. URL: https://www.vwvortex.com/threads/dsg-shift-time.3311040/ (visited on 10/08/2023).

[23] PG EmBrAAC. *Projektgruppe Emergency Braking Assistant for fully Autonomous Cars*. Sept. 2019.

[24] Azim Eskandarian, ed. *Handbook of Intelligent Vehicles*. en. London: Springer London, 2012. ISBN: 9780857290847 9780857290854. DOI: 10.1007/978-0-85729-085-4. URL: http://link.springer.com/10.1007/978-0-85729-085-4 (visited on 03/13/2024).

[25] Nikolay Falaleev. *Bird's Eye View Transformation*. URL: https://nikolasent.github.io/opencv/2017/05/07/Bird's-Eye-View-Transformation.html (visited on 05/07/2017).

[26] Brian Fitzgerald and Klaas-Jan Stol. "Continuous software engineering: A roadmap and agenda". en. In: *Journal of Systems and Software* 123 (Jan. 2017), pp. 176–189. ISSN: 01641212. DOI: 10.1016/j.jss.2015.06.063. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121215001430 (visited on 10/07/2023).

[27] ROS 2 Real-Time Working Group. *Raspberry Pi image with ROS 2 and the real-time kernel*. 2023. URL: https://github.com/ros-realtime/ros-realtime-rpi4-image (visited on 10/04/2023).

[28] Gurobi Optimization LLC. *Python API Overview - Gurobi Optimization*. 2022. URL: https://www.gurobi.com/documentation/current/refman/py_python_api_overview.html (visited on 11/23/2023).

[29] HarunTeper. *Autonomous Navigation System Simulator*. Dec. 2023. URL: https://github.com/HarunTeper/AuNa.

[30] ibaiGorordo. *onnx-Ultra-Fast-Lane-Detection-Inference*. 2022. URL: https://github.com/ibaiGorordo/onnx-Ultra-Fast-Lane-Detection-Inference.

[31] International Organization for Standardization. *Intelligent transport systems — Lanekeeping assistance systems (LKAS) —Performance requirements and testprocedures*. Tech. rep. 2014. URL: https://www.iso.org/obp/ui/en/#iso:std:iso:11270:ed-1:v1:en (visited on 10/08/2023).

[32] PG iTraffic. *TurtleBot*. 2024. URL: https://gitlab.itraffic-uol.de/itraffic/turtlebot (visited on 04/03/2023).

[33] PG iTraffic. *TurtleBot 3 Image Builder*. 2023. URL: https://gitlab.itraffic-uol.de/itraffic/TurtleBot3-image-builder (visited on 10/04/2023).

[34] Philipp Fritz Jaß. *Spezifikation und Implementierung einer Platooning Funktion auf Basis der CeCar-Plattform*. Nov. 24, 2020. URL: https://www.ifaf-berlin.de/media/Masterarbeit_Philipp_Jass.pdf.

[35] Jens Jauch et al. "Recursive B-spline approximation using the Kalman filter". In: *Engineering Science and Technology, an International Journal* 20.1 (2017), pp. 28–34. ISSN: 2215-0986. DOI: https://doi.org/10.1016/j.jestch.2016.09.015. URL: https://www.sciencedirect.com/science/article/pii/S2215098616303032.

[36] Li Kai Chun. *Vehicle-CV-ADAS*. 2023. URL: https://github.com/jason-li-831202/Vehicle-CV-ADAS.

[37] R. E. Kalman. "A New Approach to Linear Filtering and Prediction Problems". en. In: *Journal of Basic Engineering* 82.1 (Mar. 1960), pp. 35–45. ISSN: 0021-9223. DOI: 10.1115/1.3662552. URL: https://asmedigitalcollection.asme.org/fluidsengineering/article/82/1/35/397706/A-New-Approach-to-Linear-Filtering-and-Prediction (visited on 03/09/2024).

[38] Chang Mook Kang, Seung-Hi Lee, and Chung Choo Chung. "Comparative evaluation of dynamic and kinematic vehicle models". In: *53rd IEEE Conference on Decision and Control*. Los Angeles, CA, USA: IEEE, Dec. 2014, pp. 648–653. ISBN: 978-1-4673-6090-6 978-1-4799-7746-8 978-1-4799-7745-1. DOI: 10.1109/CDC.2014.7039455. URL: http://ieeexplore.ieee.org/document/7039455/ (visited on 03/13/2024).

[39] kemfic. *Curved Lane Detection*. URL: https://www.hackster.io/kemfic/curved-lane-detection-34f771 (visited on 05/24/2018).

[40] KIA Motors. *Kia EV6 Manual*. Oct. 2023. URL: https://www.kia.com/content/dam/kia2/in/en/content/ev6-manual/index.html (visited on 10/28/2023).

[41] Jason Kong et al. “Kinematic and dynamic vehicle models for autonomous driving control design”. In: *2015 IEEE Intelligent Vehicles Symposium (IV)*. ISSN: 1931-0587. June 2015, pp. 1094–1099. DOI: 10.1109/IVS.2015.7225830. URL: https://ieeexplore.ieee.org/abstract/document/7225830 (visited on 03/15/2024).

[42] Kraftfahrtbundesamt. *Monatliche Neuzulassungen September 2022*. URL: https://www.kba.de/DE/Statistik/Fahrzeuge/Neuzulassungen/MonatlicheNeuzulassungen/monatl_neuzulassungen_node.html?yearFilter=2022&monthFilter=09_september (visited on 05/11/2023).

[43] Holger Krekel. *pytest Documentation*. docs.pytest.org, Oct. 2023. URL: https://buildmedia.readthedocs.org/media/pdf/pytest/latest/pytest.pdf (visited on 10/04/2023).

[44] Janis Kröger. “Optimierung und Erweiterung einer bestehenden modellprädiktiven Regelung zur Durchführung dynamischer Überholmanöver in einem autonomen Rennbetrieb”. MA thesis. Carl von Ossietzky Universität Oldenburg, 2019.

[45] Roger Labbe. *Kalman-and-Bayesian-Filters-in-Python*. https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python. 2024.

[46] Allan Y. Lee. “Performance of Four-Wheel-Steering Vehicles in Lane Change Maneuvers”. en. In: Feb. 1995, p. 950316. DOI: 10.4271/950316. URL: https://www.sae.org/content/950316/ (visited on 03/17/2024).

[47] Junyung Lee et al. “Design of a Strategy for Lane Change Assistance System”. en. In: *IFAC Proceedings Volumes* 46.21 (2013), pp. 762–767. ISSN: 14746670. DOI: 10.3182/20130904-4-JP-2042.00134. URL: https://linkinghub.elsevier.com/retrieve/pii/S147466701638466X (visited on 02/21/2024).

[48] William Levison et al. *Development of a Driver Vehicle Module (DVM) for the Interactive Highway Safety Design Model (IHSDM)*. Nov. 2007. DOI: 10.13140/RG.2.2.35982.05446.

[49] M. Likhita et al. “Obstacle Detection in Autonomous Vehicles Using 3D LiDAR Point Cloud Data”. In: *Data Intelligence and Cognitive Informatics*. Ed. by I. Jeena Jacob, Selvanayaki Kolandapalayam Shanmugam, and Robert Bestak. Singapore: Springer Nature Singapore, 2022, pp. 745–757. ISBN: 978-981-16-6460-1.

[50] King Hann Lim, Kah Seng, and Li-Minn Ang. “River Flow Lane Detection and Kalman Filtering-Based B-Spline Lane Tracking”. In: *International Journal of Vehicular Technology* 2012 (Nov. 2012). DOI: 10.1155/2012/465819.

[51] Fernando Macedo et al. *Python StateMachine*. 2023. URL: https://python-statemachine.readthedocs.io/ (visited on 02/24/2024).

[52] Steven Macenski et al. “Robot Operating System 2: Design, architecture, and uses in the wild”. In: *Science Robotics* 7.66 (May 2022). ISSN: 2470-9476. DOI: 10.1126/scirobotics.abm6074. URL: http://dx.doi.org/10.1126/scirobotics.abm6074.

[53] Baurzhan Muftakhidinov Mark Mitchell and Tobias Winchen et al. *Engauge Digitizer Software*. URL: http://markummitchell.github.io/engauge-digitizer (visited on 10/08/2023).

[54] Felipe N. Martins, Mário Sarcinelli-Filho, and Ricardo Carelli. "A Velocity-Based Dynamic Model and Its Properties for Differential Drive Mobile Robots". en. In: *Journal of Intelligent & Robotic Systems* 85.2 (Feb. 2017), pp. 277–292. ISSN: 0921-0296, 1573-0409. DOI: 10.1007/s10846-016-0381-9. URL: http://link.springer.com/10.1007/s10846-016-0381-9 (visited on 03/13/2024).

[55] MathWorks. *Quadratic Programming - MATLAB quadprog*. 2023. URL: https://de.mathworks.com/help/optim/ug/quadprog.html (visited on 12/06/2023).

[56] Keonhee Min and Akira Ando. "Analysis on Characteristics of Dangerous Driving Events via Recorded Data of Drive-Recorder". In: *Transportation Research Procedia* 48 (2020). Recent Advances and Emerging Issues in Transport Research – An Editorial Note for the Selected Proceedings of WCTR 2019 Mumbai, pp. 1342–1363. ISSN: 2352-1465. DOI: https://doi.org/10.1016/j.trpro.2020.08.164.

[57] Lesia Mochurad, Yaroslav Hladun, and Roman Tkachenko. "An Obstacle-Finding Approach for Autonomous Mobile Robots Using 2D LiDAR Data". In: *Big Data Cogn. Comput.* 7.1 (2023), p. 43. DOI: 10.3390/BDCC7010043. URL: https://doi.org/10.3390/bdcc7010043.

[58] *mockito-python GitHub Repository*. Oct. 2023. URL: https://github.com/kaste/mockito-python (visited on 10/04/2023).

[59] Akos Nagy, Gabor Csorvasi, and Domokos Kiss. "Path planning and control of differential and car-like robots in narrow environments". en. In: *2015 IEEE 13th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*. Herl'any, Slovakia: IEEE, Jan. 2015, pp. 103–108. ISBN: 978-1-4799-8221-9. DOI: 10.1109/SAMI.2015.7061856. URL: http://ieeexplore.ieee.org/document/7061856/ (visited on 03/15/2024).

[60] Felipe Nascimento Martins and Alexandre Santos Brandão. "Motion Control and Velocity-Based Dynamic Compensation for Mobile Robots". en. In: *Applications of Mobile Robots*. Ed. by Efren Gorrostieta Hurtado. IntechOpen, Mar. 2019. ISBN: 978-1-78985-755-9 978-1-78985-756-6. DOI: 10.5772/intechopen.79397. URL: https://www.intechopen.com/books/applications-of-mobile-robots/motion-control-and-velocity-based-dynamic-compensation-for-mobile-robots (visited on 03/16/2024).

[61] OpenCV. *ArUco marker detection*. 2023. URL: https://docs.opencv.org/4.x/d9/d6d/tutorial_table_of_content_aruco.html (visited on 12/15/2023).

[62] Yan Peng et al. "The obstacle detection and obstacle avoidance algorithm based on 2-D lidar". In: *IEEE International Conference on Information and Automation, ICIA 2015, Lijiang, China, August 8-10, 2015*. IEEE, 2015, pp. 1648–1653. DOI: 10.1109/ICINFA.2015.7279550. URL: https://doi.org/10.1109/ICInfA.2015.7279550.

[63] Quang-Cuong Pham. "Trajectory Planning". en. In: *Handbook of Manufacturing Engineering and Technology*. Ed. by Andrew Y. C. Nee. London: Springer London, 2015, pp. 1873–1887. ISBN: 978-1-4471-4669-8 978-1-4471-4670-4. DOI: 10.1007/978-1-4471-4670-4_92. URL: https://link.springer.com/10.1007/978-1-4471-4670-4_92 (visited on 10/04/2023).

[64] PINTO0309. *PINTO Model Zoo*. 2023. URL: https://github.com/PINTO0309/PINTO_model_zoo/tree/main/324_Ultra-Fast-Lane-Detection-v2.

[65] Joshwa Pohlmann et al. "ROS2-based Small-Scale Development Platform for CCAM Research Demonstrators". In: *2022 IEEE 95th Vehicular Technology Conference: (VTC2022-Spring)*. 2022 IEEE 95th Vehicular Technology Conference (VTC2022-Spring). Helsinki, Finland: IEEE, June 2022, pp. 1–6. ISBN: 978-1-66548-243-1. DOI: 10.1109/VTC2022-Spring54318.2022.9860981. URL: https://ieeexplore.ieee.org/document/9860981/ (visited on 12/25/2023).

[66] Philip Polack et al. "The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles?" In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, pp. 812–818. DOI: 10.1109/IVS.2017.7995816.

[67] Philip Polack et al. "The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles?" In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. Los Angeles, CA, USA: IEEE, June 2017, pp. 812–818. ISBN: 978-1-5090-4804-5. DOI: 10.1109/IVS.2017.7995816. URL: http://ieeexplore.ieee.org/document/7995816/ (visited on 03/13/2024).

[68] Zequn Qin, Huanyu Wang, and Xi Li. *Ultra Fast Structure-aware Deep Lane Detection*. 2020.

[69] Zequn Qin, Pengyi Zhang, and Xi Li. "Ultra Fast Deep Lane Detection With Hybrid Anchor Driven Ordinal Classification". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022), pp. 1–14. DOI: 10.1109/TPAMI.2022.3182097.

[70] *Raspberry Pi - Camera 3*. URL: https://www.raspberrypi.com/products/camera-module-3/ (visited on 04/01/2024).

[71] *Raspberry Pi 4 Model B*. URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/ (visited on 04/01/2024).

[72] Robert Bosch GmbH. *Lane Keeping Assist*. 2023. URL: https://www.bosch-mobility.com/en/solutions/assistance-systems/lane-keeping-assist/ (visited on 10/31/2023).

[73] Robotis. *Robotis TurtleBot 3 e-Manual Chapter Two*. Online. Nov. 2023. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/features/.

[74] *Ruff Python Linter GitHub Repository*. Oct. 2023. URL: https://github.com/astral-sh/ruff (visited on 10/07/2023).

[75] J.Z. Sasiadek. “Sensor fusion”. In: *Annual Reviews in Control* 26.2 (2002), pp. 203–228. ISSN: 1367-5788. DOI: https://doi.org/10.1016/S1367-5788(02)00045-7. URL: https://www.sciencedirect.com/science/article/pii/S1367578802000457.

[76] Dieter Schramm, Manfred Hiller, and Roberto Bardini. *Vehicle Dynamics*. en. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018. ISBN: 978-3-662-54482-2 978-3-662-54483-9. DOI: 10.1007/978-3-662-54483-9. URL: http://link.springer.com/10.1007/978-3-662-54483-9 (visited on 03/13/2024).

[77] Scrum.org. *What is Scrum?* 2023. URL: https://www.scrum.org/learning-series/what-is-scrum (visited on 10/08/2023).

[78] Rahul Sharma K., Daniel Honc, and Frantisek Dusek. “Predictive Control Of Differential Drive Mobile Robot Considering Dynamics And Kinematics”. en. In: *ECMS 2016 Proceedings edited by Thorsten Claus, Frank Herrmann, Michael Manitz, Oliver Rose*. ECMS, June 2016, pp. 354–360. ISBN: 978-0-9932440-2-5. DOI: 10.7148/2016-0354. URL: http://www.scs-europe.net/dlib/2016/2016-0354.htm (visited on 03/13/2024).

[79] *Shift points on Mk7 TDI manual?* URL: https://forums.tdiclub.com/index.php?threads/shift-points-on-mk7-tdi-manual.431653/ (visited on 04/03/2024).

[80] Michał Siwek et al. “Identification of Differential Drive Robot Dynamic Model Parameters”. en. In: *Materials* 16.2 (Jan. 2023), p. 683. ISSN: 1996-1944. DOI: 10.3390/ma16020683. URL: https://www.mdpi.com/1996-1944/16/2/683 (visited on 03/13/2024).

[81] Riikka Soitinaho, Marcel Moll, and Timo Oksanen. “2D LiDAR based object detection and tracking on a moving vehicle”. In: *IFAC-PapersOnLine* 55.32 (2022). 7th IFAC Conference on Sensing, Control and Automation Technologies for Agriculture AGRICONTROL 2022, pp. 66–71. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2022.11.116. URL: https://www.sciencedirect.com/science/article/pii/S2405896322027495.

[82] Qunying Song, Emelie Engström, and Per Runeson. “Concepts in Testing of Autonomous Systems: Academic Literature and Industry Practice”. In: *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*. 2021, pp. 74–81. DOI: 10.1109/WAIN52551.2021.00018.

[83] Manideep Sridhara. *TuSimple - Ace the Lane Detection Challenge*. 2021. URL: https://www.kaggle.com/datasets/manideep1108/tusimple.

[84] Anthony Stark. *Vehicle acceleration and maximum speed modeling and simulation*. 2022. URL: https://x-engineer.org/vehicle-acceleration-maximum-speed-modeling-simulation/ (visited on 11/16/2023).

[85] Forschungsgesellschaft für Straßen- und Verkehrswesen, ed. *Richtlinien für die Markierung von Straßen. Teil A: Autobahnen*. ger. Ausgabe 2019. FGSV 330A. Cologne: Forschungsgesellschaft für Straßen- und Verkehrswesen e.V, 2019. ISBN: 978-3-86446-251-1.

[86] *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. Apr. 2021. URL: https://www.sae.org/standards/content/j3016_202104/ (visited on 10/05/2023).

[87] *Technische Daten aller VW Golf 7 Modelle*. URL: https://carwiki.de/vw-golf-7-technische-daten/ (visited on 04/03/2024).

[88] Tino Teige et al. "Two Decades of Formal Methods in Industrial Products at BTC Embedded Systems". In: *Formal Methods*. Ed. by Marieke Huisman, Corina Păsăreanu, and Naijun Zhan. Cham: Springer International Publishing, 2021, pp. 725–729. ISBN: 978-3-030-90870-6.

[89] *The 25 Top Causes of Car Accidents in the US*. Jan. 9, 2024. URL: https://web.archive.org/web/20240229163317/https://seriousaccidents.com/legal-advice/top-causes-of-car-accidents/.

[90] The Robotis authors. *LDS-02*. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_02/ (visited on 04/01/2024).

[91] The Robotis authors. *OpenCR 1.0*. URL: https://emanual.robotis.com/docs/en/parts/controller/opencr10/ (visited on 04/01/2024).

[92] *Tire and Wheel Plus Sizing - Tire Size Calculator*. URL: https://www.1010tires.com/Tools/Tire-Size-Calculator/205-55R16?active=0&ismetric=true (visited on 04/03/2024).

[93] *TurtleBot3*. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/ (visited on 04/01/2024).

[94] *Vehicle acceleration and maximum speed modeling and simulation*. URL: https://x-engineer.org/vehicle-acceleration-maximum-speed-modeling-simulation/ (visited on 10/08/2023).

[95] Guojun Wang et al. "A Comprehensive Testing and Evaluation Approach for Autonomous Vehicles". In: *WCX World Congress Experience*. SAE International, Apr. 2018. DOI: https://doi.org/10.4271/2018-01-0124. URL: https://doi.org/10.4271/2018-01-0124.

[96] Nathan Wies. *Multicast over Wireless*. May 2, 2019. URL: https://wirelesslywired.com/2019/05/02/multicast-over-wireless/.

[97] Lukas Wunderli. *MPC based Trajectory Tracking for 1:43 scale Race Cars*. Tech. rep. Automatic Control Laboratory (IfA), Swiss Federal Institute of Technology (ETH) Zurich, Apr. 2011.

[98] Gao Zhenhai et al. "Multi-argument Control Mode Switching Strategy for Adaptive Cruise Control System". In: *Procedia Engineering* 137 (2016). Green Intelligent Transportation System and Safety, pp. 581–589. ISSN: 1877-7058. DOI: https://doi.org/10.1016/j.proeng.2016.01.295. URL: https://www.sciencedirect.com/science/article/pii/S1877705816003222.